electronics

MDPI

Article

Synchronized Delay Measurement of Multi-Stream Analysis over Data Concentrator Units

Anvarjon Yusupov, Sun Park * and JongWon Kim

AI Graduate School, Gwangju Institute of Science and Technology (GIST), Gwangju 61005, Republic of Korea; ayusupov@gm.gist.ac.kr (A.Y.); jongwon@gist.ac.kr (J.K.)
* Correspondence: sunpark@gist.ac.kr

Abstract: Autonomous vehicles (AVs) rely heavily on multi-modal sensors to perceive their surroundings and make real-time decisions. However, the increasing complexity of these sensors, combined with the computational demands of AI models and the challenges of synchronizing data across multiple inputs, presents significant obstacles for AV systems. These challenges of the AV domain often lead to performance latency, resulting in delayed decision-making, causing major traffic accidents. The data concentrator unit (DCU) concept addresses these issues by optimizing data pipelines and implementing intelligent control mechanisms to process sensor data efficiently. Identifying and addressing bottlenecks that contribute to latency can enhance system performance, reducing the need for costly hardware upgrades or advanced AI models. This paper introduces a delay measurement tool for multi-node analysis, enabling synchronized monitoring of data pipelines across connected hardware platforms, such as clock-synchronized DCUs. The proposed tool traces the execution flow of software applications and assesses time delays at various stages of the data pipeline in clock-synchronized hardware. The various stages are represented with intuitive graphical visualization, simplifying the identification of performance bottlenecks.

Keywords: delay measurement; connected autonomous vehicles; DCU; temporal synchronization; multi-stream analysis; runtime visualization

Academic Editor: Arman Sargolzaei

Received: 29 November 2024
Revised: 20 December 2024
Accepted: 24 December 2024
Published: 27 December 2024

Citation: Yusupov, A.; Park, S.; Kim, J. Synchronized Delay Measurement of Multi-Stream Analysis over Data Concentrator Units. *Electronics* **2025**, *14*, 81. https://doi.org/10.3390/electronics14010081

1. Introduction

Nowadays, millions of self-driving cars are operating around the globe, which decreases the number of traffic accidents substantially [1]. The modern generation of autonomous vehicles (AVs) enables driving automation that substitutes human drivers to control vehicles with superior perception, recognition, decision-making, and driving skills. Perception information in AVs relies on using one or more sensor modules, such as camera, radar, and light detection and ranging (LiDAR) [2]. Although it is possible to take advantage of these sensors in terms of providing a robust and complete description of the surrounding area, there are still various open problems related to multiple sensor operation, including increased failure of detection, data synchronization, adverse weather, etc. V2X communication techniques are also mentioned, which are specially designed for connected AVs [3]. As AVs rely on various sensors to understand their environment, the sheer volume of data generated by multiple sensors poses a considerable computational burden [4]. The concept of DCUs (data concentrator units) [5] is being studied to efficiently process various sensor data collected in autonomous vehicles. DCUs play a significant role in intelligent decision-making of AVs. By acting as information hubs, DCUs collect and process data from various sensors, including cameras, LiDAR systems, radars, and ultrasonic detectors.

As the number of sensors increases, the demand on onboard computing systems intensifies, leading to delays in real-time decision-making and responsiveness. This becomes particularly problematic in complex environments, where quick reactions are essential for safety [3]. Additionally, managing sensor fusion, where data from multiple sources such as cameras, LiDAR, and radar are combined, becomes more intricate as the sensor array grows. One of the main challenges of increasing the quantity of sensors is developing an accurate system that is capable of extracting all of the relevant data from each sensor while minimizing potential errors [6]. Although multiple sensors' input increases safety through redundant data, there are still challenges in increased continuous sensory inputs because of various constraints coming from the implementation and cost of those sensors [7]. The overall cost and complexity of the autonomous vehicle system escalate as more sensors are added, which could slow down the widespread adoption and affordability of autonomous technology.

Another issue is that aligning and synchronizing data from various sensors accurately becomes a daunting task, and any synchronization errors can lead to inaccurate or incomplete environmental understanding (i.e., the vehicle's external environment). As Liangkai Liu et al. states, multi-sensor data synchronization is still one of the open challenges in AV development [3]. In the realm of AVs, sensor synchronization emerges as an important architectural element essential for achieving accurate and coherent perception. As AVs navigate complex environments, data from diverse sensors must harmoniously align to create a comprehensive and unified representation of the surroundings [4]. This synchronization ensures temporal consistency and enhances the AV's ability to navigate safely and autonomously by providing a reliable foundation for perceiving dynamic scenarios and responding with precision to changing road conditions [8].

To summarize, this study addresses three primary challenges in AV systems: the growing complexity of sensor systems, increasing computational demands, and data synchronization issues. While upgrading hardware or deploying advanced AI models may improve performance, these approaches are costly and not always practical. Instead, optimizing the current system by identifying bottlenecks within the data pipeline, from sensor data collection to processing, can enhance system responsiveness and accuracy more cost-effectively. Numerous studies have tackled these issues by focusing on profiling, runtime verification, and real-time monitoring tools. Profiling tools such as Callgrind [9], DTrace [10], DCU-CHK [11], and Autoware_Perf [12] help identify inefficiencies in CPU and memory usage but are typically limited to single-machine environments and introduce significant overhead. Runtime verification tools, including Java PathExplorer (JPAX) [13], JavaMOP [14], Bro/Zeek [15], BusMOP [16], ROS 2 Tracing [17], Las-Casas's Anomalous Trace Detection [18], ensure system correctness but emphasize fault detection over performance optimization. Real-time monitoring tools such as LTTng [19], Tigris Framework [20], Software Runtime Monitoring Tool (SRMT) [21], Jhonny Mertz et al.'s Adaptive Runtime Monitoring [22]. provide low-latency performance monitoring but lack multi-node synchronization and visualization capabilities. To address these limitations, we previously proposed a delay measurement concept [23–25], to monitor and optimize real-time, multi-node systems. In this paper, we implement this conceptual method and validate it through experiments and use cases, demonstrating its potential for improving the efficiency of AV data pipelines.

This paper proposes a delay measurement tool to analyze the performance of AI-powered data pipelines in multiple connected DCUs. Designed for multi-node environments, the tool synchronizes performance metrics across distributed hardware units, enabling comprehensive delay analysis across distributed systems in real time. The tool comprises three components: a software tracing module to capture execution flow and

runtime behavior, a delay calculation module to quantify and analyze latency in data pipelines, and a delay visualization module to graphically represent latency, enabling intuitive identification of performance bottlenecks. The tool is designed to reduce the overhead of software tracing, enable precise latency measurement, and provide real-time visualization across connected nodes. In our use case, the tool successfully measured and visualized delays in sensor data processing across multiple DCUs, ensuring synchronized and accurate delay analysis. Temporal synchronization aligns performance metrics across distributed hardware, enabling accurate latency tracking and addressing even minor delays. These capabilities aim to improve the efficiency and responsiveness of AI data pipelines by enabling better performance monitoring, facilitating the identification of bottlenecks, and supporting more informed real-time decision-making.

The remainder of the paper is organized as follows. Section 2 describes the related studies on the delay measurements and DCUs; Section 3 presents the proposed delay measurement tool for multi-stream analysis. Then, Section 4 shows the experimental results, and Section 5 concludes the paper.

2. Related Works

2.1. Delay Measurement

In this subsection, we will describe and explain in detail the related works of delay measurements, which are divided into three categories: profiling, runtime verification, and real-time monitoring. The profiling tools are to profile the system performance and detect the bottleneck. Valgrind/Callgrind is widely used for detailed memory profiling and function call tracking, helping developers identify bottlenecks by simulating cache behavior and tracing execution paths [9]. Its advantage lies in providing in-depth insights into memory management, though it suffers from significant performance overhead, which limits its suitability for real-time systems. DTrace offers a dynamic tracing framework for both kernel and user-level applications, allowing for real-time diagnostics without requiring a system restart [10]. However, it lacks support for distributed system tracing and is predominantly used for single-system environments. DCU-CHK focuses on detecting bottlenecks in large-scale CPU-DCU systems, offering a more specialized approach for AV environments [11], but it is limited to high-level checkpointing and may not offer detailed bottleneck analysis. Lastly, Autoware_Perf addresses performance bottlenecks in ROS 2-based AV systems, particularly by monitoring communication latency and execution delays [12]. Its disadvantage is its specific focus on ROS 2 systems, limiting its generalizability to other autonomous systems. These tools provide in-depth performance insights, helping to optimize system resources and identify inefficiencies. But they are limited to single-machine environments and can introduce performance overhead, making them less effective for real-time, multi-node, sensor-heavy environments like AVs.

The runtime verification ensures system correctness and detection of anomalies by verifying software behavior during runtime. The JPAX monitors Java applications for concurrency issues and data races through runtime verification techniques, ensuring system correctness in multithreaded environments [13]. However, its primary limitation is that it is Java-specific, reducing its applicability in non-Java systems. JavaMOP allows developers to specify properties that must hold during execution, offering real-time feedback on any violations [14], but its reliance on aspect-oriented programming (AOP) increases complexity in system integration. Bro/Zeek focuses on real-time network traffic monitoring, detecting security anomalies [15], though it is less suited for general-purpose runtime verification. BusMOP applies runtime verification to PCI peripherals, monitoring hardware behavior to detect faults [16], but its hardware focus limits its flexibility for broader applications. ROS 2 tracing provides detailed tracing of callbacks and communication events in ROS

2-based systems, offering valuable insights for AV applications [17], but its scope is limited to ROS 2 environments. Finally, Las-Casas's anomalous trace detection provides a method for capturing outliers in trace logs, enhancing system fault detection [18], though its focus on anomaly detection may not cover broader performance issues. These tools ensure robust system correctness, detecting outliers and faults that could compromise system integrity, which is essential for safety-critical systems like AVs. However, they tend to be domain-specific and are more focused on correctness than on real-time performance optimization or data handling across distributed systems, which is critical in autonomous systems.

Real-time monitoring ensures real-time performance monitoring while using adaptive sampling and monitoring techniques to minimize system overhead. The LTTng provides low-overhead tracing in Linux environments, offering detailed performance monitoring with minimal resource consumption [19], though it does not focus on multi-node delay measurement in distributed systems. The Tigris Framework introduces goal-specific adaptive monitoring, dynamically adjusting the sampling rate based on performance goals [20], but this also means that less granular data may be collected. SRMT provides dynamic runtime monitoring for C/C++ applications, offering adaptive monitoring to reduce performance overhead [21], though it is aimed only for C/C++ applications and was mainly tested in Windows OS. Jhonny Mertz et al. propose a dynamically adjustable sampling rate based on system workload, optimizing real-time performance [22], but the approach lacks real-time visualization capabilities and synchronization across multiple nodes. These tools ensure low-latency performance by dynamically adjusting to system conditions, making them useful for distributed, real-time systems. However, they may lack multi-node synchronization and advanced visualization features, limiting their ability to detect detailed bottlenecks and synchronize data across distributed nodes.

The comparative analysis presented in Table 1 highlights the key capabilities of the proposed delay measurement tool in the context of existing solutions. Unlike profiling tools such as Valgrind, DTrace, and DCU-CHK, which focus on memory tracking, function call profiling, or specific system checkpoints, the proposed tool provides comprehensive multi-node delay measurement and visualization capabilities. This specialization is critical for distributed systems where real-time performance and synchronization between hardware units are essential. Similarly, runtime verification tools like JPAX, JavaMOP, and Bro/Zeek, while effective in identifying anomalies or ensuring system properties, lack the scalability and real-time analysis features necessary for handling high-throughput video and sensor data.

A distinguishing feature of the proposed tool is its support for multi-node visualization and synchronization, enabling delay measurement across multiple streams in real-time scenarios. This is particularly important for applications requiring synchronized operations across distributed hardware, such as autonomous systems and sensor networks. In contrast, real-time monitoring tools like LTTng and SRMT, while capable of low-overhead tracing or dynamic monitoring, fall short in handling multiple nodes or providing interactive visualization. The proposed tool addresses these gaps by integrating real-time delay analysis with advanced visualization capabilities, making it a robust solution for optimizing system performance in latency-sensitive environments. Our tool leverages and modifies certain capabilities of LTTng to enable multi-node analysis. The findings emphasize the tool's unique position in advancing the field of delay measurement and real-time monitoring.

Table 1. Comparative analysis of proposed tool and related work.

Tool	Category	Strengths	Limitations	Multi-Node Visualization	Multi-Node Analysis	Synchronization
Valgrind /Callgrind	Profiling	Detailed memory profiling and function call tracking	High performance overhead	No	No	No
DTrace	Profiling	Real-time diagnostics without requiring system restart	Limited to single-system environments	No	No	No
DCU-CHK	Profiling	Specialized for AV environments	High-level checkpointing, lacks detailed analysis	No	No	No
Autoware_Perf	Profiling	Monitors communication latency in ROS 2 systems	Specific to ROS 2 systems	No	No	No
JPAX	Runtime verification	Concurrency issue detection in Java applications	Java-specific, limited applicability	No	No	No
JavaMOP	Runtime verification	Real-time feedback on property violations	Complexity in system integration	No	No	No
Bro/Zeek	Runtime verification	Real-time network traffic monitoring	Focuses on network traffic only	No	No	No
BusMOP	Runtime verification	Monitors PCI peripherals for faults	Limited to hardware monitoring	No	No	No
ROS 2 tracing	Runtime verification	Detailed tracing of callbacks in ROS 2 systems	Scope limited to ROS 2	No	No	No
Las-Casas's anomalous trace detection	Runtime verification	Detects outliers in trace logs	Focused on anomaly detection only	No	No	No
LTTng	Real-time monitoring	Low-overhead tracing in Linux environments	Lacks multi-node delay measurement	No	No	No
Tigris Framework	Real-time monitoring	Adaptive monitoring with goal-specific sampling	Granularity trade-offs in adaptive sampling	No	No	No
SRMT	Real-time monitoring	Dynamic monitoring for C/C++ applications	Limited to Windows OS, lacks visualization	No	No	Partial
Proposed delay measurement tool	Multi-module delay analysis	Multi-node synchronization and visualization	Requires hardware synchronization; domain-specific	Yes	Yes	Yes

2.2. Data Concentrator Unit

The data concentrator unit (DCU) in our research plays an essential role in managing, processing, and analyzing data flows from various sensors inside autonomous vehicles (AVs). Acting as information hubs, DCUs collect and process data from various sensors, including cameras, LiDAR systems, radar, and ultrasonic detectors. Such functionalities are similar to one of the key functionalities of Domain Controllers in AVs. The transition from traditional electronic control units (ECUs) to domain controllers is being increasingly adopted in modern AV architectures to handle the complex data flows generated by sensors such as LiDAR, radar, and cameras. Domain controllers, just like our DCUs, serve as centralized hubs to process and fuse sensor data in real time [26].

The perception module in the vehicle design proposed by Wenfu Wang et al. [27] exhibits functionalities similar to our DCUs, particularly in its object-detection capabilities. Specifically, our DCU is designed to collect synchronized frames from multiple cameras and perform object-detection analysis. This approach aligns with the concept of a domain

controller in AVs as described in the domain-based architecture by Onur Alparslan et al. [28], where data collection and processing from multiple sensors are key functionalities. Our DCU acts as a domain hub for multiple video sensors. Additionally, the trend towards connecting multiple domain controllers to an Electronic System Unit (ESU) via topology zones, as discussed in Alparslan's work [28], is becoming more popular and is predicted to be a standard in the future. In our research, we also test the proposed tool across multiple DCUs connected to sensors, a common architecture in modern AVs, as highlighted in the paper by Arya G. Nair et al. [29]. Sensor synchronization, a vital component of our design, is also emphasized as an important aspect in autonomous vehicle design by Gustavo Velasco-Hernandez et al. [30]. Furthermore, a software-based sensor synchronization solution is presented by Hang Hu et al. [31]; however, our design integrates both software and hardware synchronization, where the DCUs are synchronized via software and cameras via hardware.

In our previous paper [5,25], we proposed a design for DCUs with synchronized cameras. However, in the current paper, the DCU setup is enhanced using hardware clock synchronization among DCUs. The added hardware clock synchronization ensures that all devices in a multi-hardware system operate on a consistent time reference, which is important for accurate delay measurement. By synchronizing clocks across different hardware, the proposed delay measurement tool can precisely compare timestamps from various sources, allowing for reliable calculation of delays between processes occurring on different devices. This synchronization is essential for maintaining the integrity and accuracy of the delay data, particularly in distributed systems, where timing discrepancies could otherwise lead to incorrect or inconsistent results. The DCU initially was a part of our previous works [24,25], where we proposed the Conceptual AI-integrated V2X-Car Edge Cloud, as shown in Figure 1, which can efficiently manage an edge cloud for V2X-car services that can provide efficient cloud services based on large amounts of V2X communication information. The legend in Figure 1 shows the connection ports of each hardware in the diagram. The figure's hardware can be divided as follows: AI+X Post, K8S Worker for AI Computing and Storage, K8S Worker for Data IO, Hybrid-V2X Mobile Station, SmartX Pole, SiLS with PreScan. The delay measurement tool was applied to intuitively understand the process of the internal analysis model between the computing and storage worker, data IO worker, and SiLS of the V2X-car edge cloud.

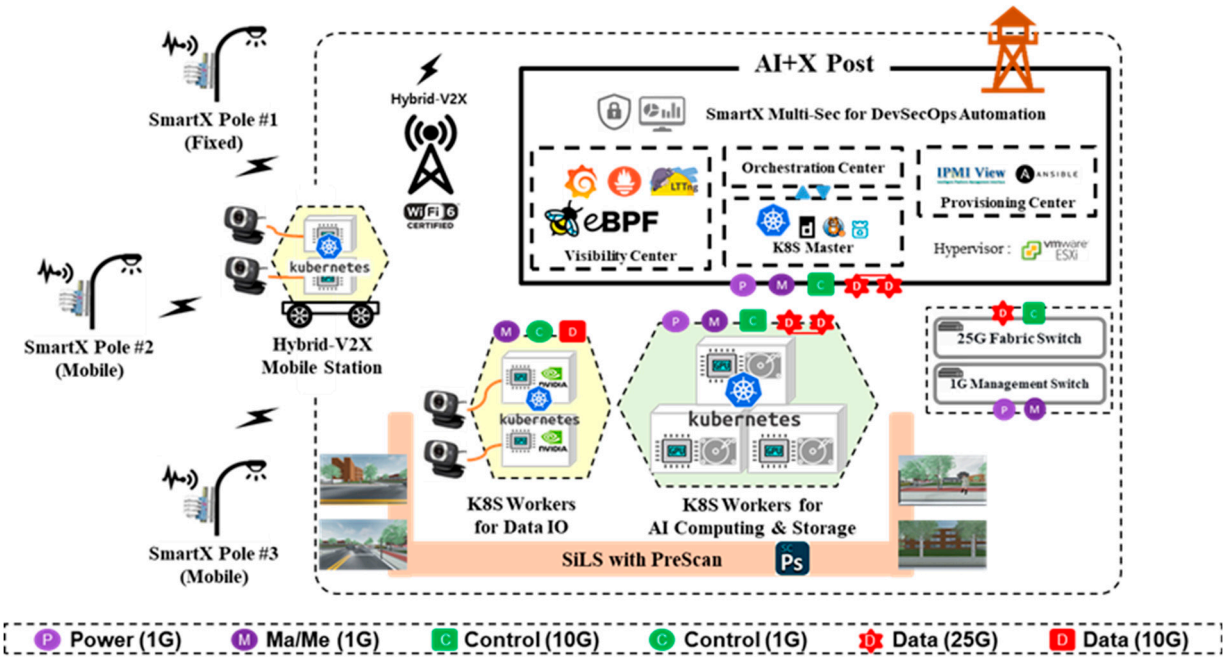

Figure 1. Conceptual diagram of AI-integrated V2X-Car Edge Cloud.

3. Delay Measurement Tool

This section describes the design and implementation of the proposed delay measurement tool. Figure 2 shows the conceptual diagram of the delay measurement tool and presents the key functionalities of the three software modules: software tracing in Figure 2a, delay calculations in Figure 2b, and delay visualization in Figure 2c. Additionally, Figure 2 shows the hardware unit distribution for operating our software modules. The hardware units are labeled "target system" and "monitoring system." These system terms will be used often in this paper to define the role of hardware in operation design for the proposed delay measurement tool. "Target systems" refers to one or many pieces of hardware, monitored by our proposed tool (i.e., monitored for delay measurement). The "monitoring system" term describes hardware that overviews multiple target systems, creating a centralized latency analyzing unit. We designed the tool to have one single monitoring system that overviews various target systems.

The software tracing module in Figure 2a runs on the target system, while the DCU (i.e., target system) performs the sensor's data analysis. The software tracing module is a significant component that collects trace data that contain a comprehensive view of the program's behavior and performance. The delay calculation in Figure 2b and delay visualization modules in Figure 2c both run directly in the monitoring system. The delay calculation module collects trace data from multiple target systems and calculates the runtime for each target hardware. After calculations, the delay calculation module performs pre-processing and sends the result of the calculations to the delay visualization module. The delay visualization module runs the GUI-based representation of the delay for the sensor's data analysis on DCUs. The visualization updates constantly as the delay calculation module sends the delay data. The visualization module allows us to visualize the delay measurement of data analysis from the multiple DCUs using a user-friendly graphical interface. The GUI allows us to compare and observe the performance of several data analyses from the DCUs simultaneously. In further subsections, we present a detailed description of the software module with a deeper explanation of the functionalities of our proposed delay measurement software tool.

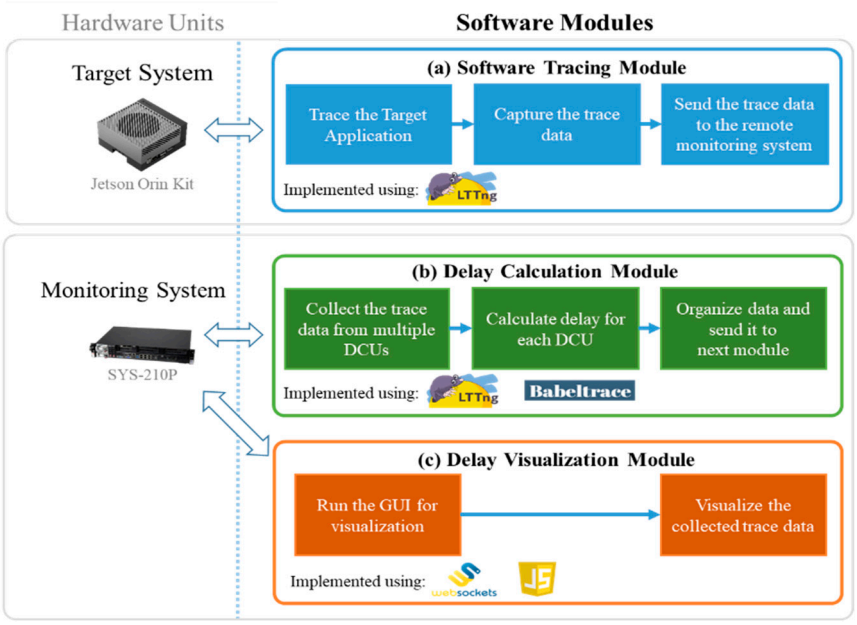

Figure 2. Conceptual diagram of the delay measurement tool.

3.1. Software Tracing Module

The software tracing module is the first module that is executed by the delay measurement software tool. The primary purpose of the module is to gather the information on the performance of the data analysis on DCUs. The software tracing module includes the insertion of specialized code, known as tracepoints, at strategic locations within the codebase to collect detailed information about the program's execution runtime. These tracepoints act as markers that, when triggered, generate trace data containing a timestamp, the codebase location, and other relevant details. Thus, the main objective of the software tracing module is to generate accurate trace data. The trace data offer insights into the runtime behavior of the sensor's data analysis, enabling us to identify bottlenecks in the software. Our proposed tool is aimed at identifying latency during data analysis. The proposed tool can be applied on any sort of analysis. However, having intentions to test our tool in practice, we pick a specific sensor data analysis for testing. In this paper, we used the object-detection analysis use case as an example to show the interaction between each module of our proposed delay measurement software tool. The use case of data analysis is object-detection analysis over multiple synchronized video streams. In this subsection, we explain the implementation of the software tracing module. Our proposed software tracing module is implemented by leveraging and modifying certain capabilities of the LTTng version 2.12 [19] and the Nvidia DeepStream software version 6.2 [32] to show internal function interactions and architecture, as shown in Figure 3. LTTng (Linux Trace Toolkit: next generation) was chosen as the primary tracing tool for this study due to its scalability, low overhead, and robust support for both live and offline analysis. In contrast to alternative tracing tools such as SystemTap or strace, which often introduce significant performance degradation during operation, LTTng provides high-efficiency tracing capabilities, making it particularly suitable for real-time environments where low latency and minimal interference are critical. Additionally, LTTng's ability to capture detailed performance metrics across both kernel and user-space levels proved invaluable for instrumenting the DeepStream pipelines. This allowed for comprehensive monitoring of runtime performance, including critical events at various stages of the pipeline.

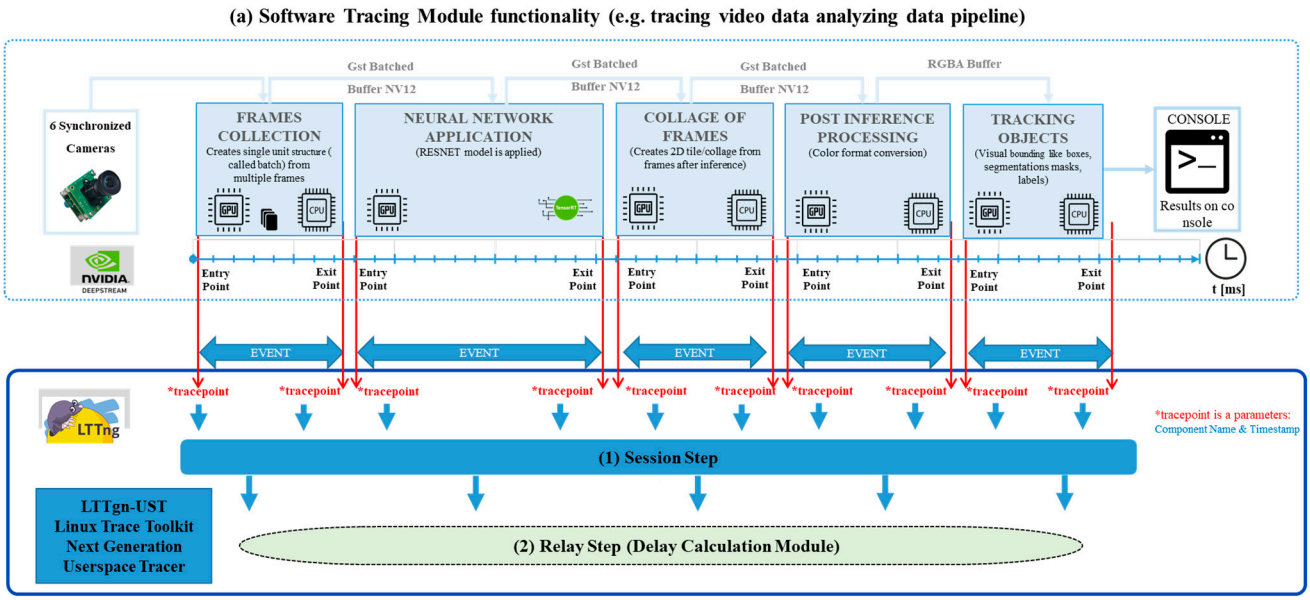

Figure 3. Design of software tracing module.

The integration of LTTng with Babeltrace further enhanced its utility, facilitating seamless trace data parsing and enabling the efficient transformation of raw trace events into structured, interpretable formats. This feature was essential for calculating delays, identifying performance bottlenecks, and visualizing system behavior in a distributed environment. By leveraging these capabilities, we were able to implement a highly granular tracing framework that provided insights into the timing and synchronization of processes across multiple nodes. To adapt LTTng for multi-node setups involving data concentrator units (DCUs), modifications were introduced to its relay mechanism, enabling synchronized tracing across distributed hardware. These enhancements ensured that trace data from multiple nodes could be gathered concurrently while maintaining temporal alignment, a critical requirement for analyzing delays in heterogeneous hardware architectures. This adaptation effectively bridged the gap between traditional single-node tracing and the requirements of multi-node environments, aligning the tool's functionality with the real-time demands of autonomous systems and other distributed infrastructures.

Moreover, LTTng's inherent scalability allowed for its deployment across various hardware configurations, providing a robust foundation for future expansion to larger, more complex multi-node setups. These characteristics underscore its suitability as the backbone of the delay measurement tool and highlight its advantages over alternative solutions in real-time, distributed monitoring scenarios.

The modified tracing tool (i.e., LTTng) has been integrated into DeepStream pipeline to collect trace data on the object-detection performance. The process of integration is also called instrumentation of the software. Figure 3 shows the design of the conceptual diagram of the software tracing module and includes an instrumentation diagram which consists of the functionality of the software tracing module and the architecture of the software tracing module.

Figure 3a shows the basic functionality of the software tracing module, specifically the tracing of the use case sensor's data analysis (e.g., object detection using the video data analysis pipeline). The software tracing module, as shown in Figure 3, was carefully designed to capture delays at critical points within the NVIDIA DeepStream pipeline. This module's primary objective is to gather accurate and granular trace data to analyze delays and identify bottlenecks across the data pipeline components. To achieve this, the tracing process relies on tracepoints as the core mechanism for capturing runtime execution details. Tracepoints were strategically placed at both the entry and exit points of each pipeline component, enabling precise measurement of delays introduced by the system. Specifically, the DeepStream pipeline in our implementation comprises five main components—frame collection, neural network application, collage of frames, post-inference processing, and tracking objects. As a result, a total of 10 tracepoints (2 for each component) were deployed. This design decision ensures that delays are captured both at the level of individual components and across the entire pipeline, offering fine-grained visibility into runtime performance. Each tracepoint captures two critical parameters essential for delay calculation: the component name and timestamps. The component name serves as a unique identifier for each pipeline section, enabling accurate mapping of trace data to specific points in the code. The timestamps, on the other hand, are hardware-assisted UNIX timestamps synchronized using Precision Time Protocol (PTP). PTP ensures sub-microsecond accuracy across nodes, making it a highly reliable mechanism for delay measurement in distributed systems. The choice to use hardware-assisted timestamps, rather than software-based clocks, was guided by the need to minimize software-induced overhead and ensure precise real-time performance monitoring. Delays in this study are defined as the time difference between an entry and exit tracepoint for an individual pipeline component or cumulatively as the total time required for a frame to pass through the entire pipeline. The design

decisions underlying the software tracing module—such as the placement of tracepoints, the use of hardware-assisted timestamps, and the adoption of synchronized frame rates—were made to ensure precision, modularity, and real-world applicability. This systematic approach provides the flexibility to extend the tool to other types of pipelines or distributed frameworks, further validating its utility for delay measurement and performance analysis in complex multi-node systems.

Figure 3b shows the architecture of the software tracing module, which is the LTTng (i.e., use case). The figure presents two main steps of the software tracing architecture of leveraged and modified LTTng [19]: (1) the consumer step and (2) the relay step. Both steps are part of the architecture of LTTng, and our research leverages and further modifies certain LTTng functionalities. While the consumer step retains its core functionality of extracting trace data from the kernel, the relay step is significantly enhanced to enable simultaneous software tracing across multiple hardware units. This adaptation is critical for experiments involving multiple data concentrator units (DCUs), allowing synchronized analysis of trace data from distributed systems. In the modified setup, the relay step runs on monitoring hardware, acting as a centralized hub for receiving trace data from multiple DCUs concurrently. By specifying the destination IP address of the monitoring system, trace data from each DCU is routed directly to the relay. The relay handles multiple TCP connections simultaneously, ensuring real-time synchronization and data integrity. These enhancements enable accurate delay calculations by capturing trace data from all hardware nodes in parallel. This architecture extends LTTng's capabilities, bridging the gap between single-node tracing and the demands of distributed, multi-node environments, and is particularly suited for real-time delay measurement and performance analysis.

3.2. Delay Calculation Module

The delay calculation module is designed to measure the latency of the sensor data analysis based on the trace data generated from the software tracing module in Figure 3. Delay calculation at its core is an algorithm that takes trace data as input and generates the delay data, which is information on the delay of various components of data analysis (in our use case, we leverage the components of the DeepStream pipeline). Delay calculation run on the monitoring hardware receiving trace data of multiple DCUs (i.e., target hardware). The delay calculation module is closely linked to the delay visualization module, which we will also describe in Section 3.3.

Figure 4 shows the conceptual design of the delay calculation module. The relay step Figure 4 (3) is process that constantly receives and saves all trace data sent from target systems. The relay step categorizes received trace data by the hardware name of the sender (i.e., target system). While trace data are being received by the relay step, as shown Figure 4, we launch our delay calculations and delay visualization components. Delay calculation constantly measures delay, pre-processes the result, and sends the results to the delay visualization module, which constantly updates the visualization based on the received delay data. Both modules are running in independent loops separately, but they also share data (i.e., delay calculation sends delay data to delay visualization). In order to implement such concept, we carried out the multiprocessing with the Python library. Multiprocessing in Python version 3.8.10 refers to the capability of running multiple processes concurrently to achieve parallel execution. It also allows for sharing of data between processes. In our case, the processes were delay calculation and delay visualization modules. The pipe launches both modules at the same time and creates pipe connection, through which we send the delay data.

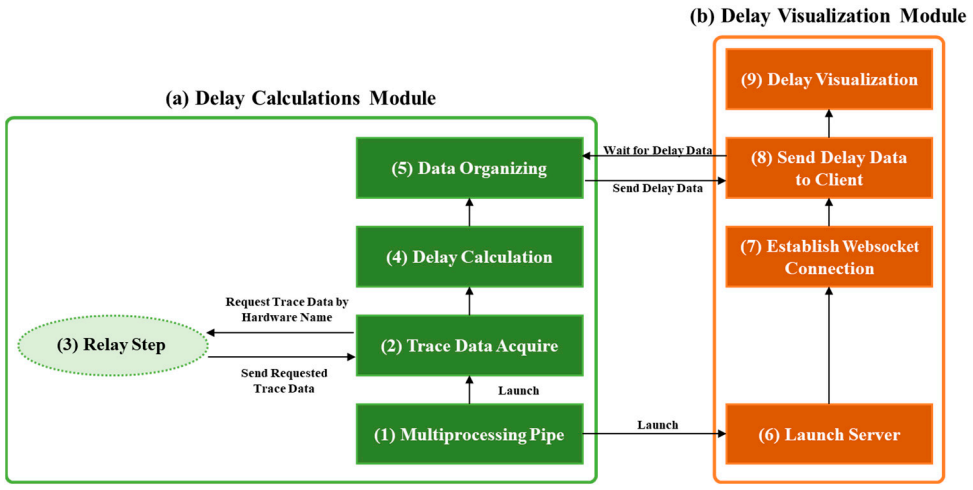

Figure 4. Processing design of delay calculation and visualization module.

As we can see in Figure 4, the delay calculation part of the module runs in the loop and performs tasks labeled (1) multiprocessing pipe, (2) trace data acquire, (3) relay step, (4) delay calculation, and (5) data organizing. First, we run the multiprocessing pipe that has been previously explained, which will launch both modules. Next, in the trace data acquire task, as in Figure 4 (2), we send requests to the relay step to send us the trace data. Each time, we request trace data by the hardware name of the target system (as already stated, the relay step categorizes trace data by the name of target system). After receiving requested trace data, we perform the extraction of trace data parameters. Such manipulations, as requested from the relay step and the extraction of trace data parameters, are implemented using the Babeltrace2 [33] Python API. The combination of Babeltrace and LTTng is a common practice for trace analysis, particularly in scenarios requiring efficient tracing of single systems. However, the approach proposed in this paper introduces a novel enhancement by enabling the simultaneous gathering of trace data from multiple hardware units. This is achieved by employing a modified relay mechanism in the LTTng architecture that concurrently requests and streams trace data from multiple sources to a centralized monitoring system. Unlike traditional setups where relay steps handle sequential or isolated traces, the proposed method ensures synchronization and reduces potential bottlenecks in multi-node environments. In the enhanced design, the LTTng relay is adapted to handle multiple concurrent TCP connections, each corresponding to an individual hardware unit. One of the keys of achieving this is Babeltrace-based message iterators. By utilizing these message iterators simultaneously, the monitoring system can aggregate trace data in real time, maintaining temporal coherence across the nodes. Babeltrace, traditionally used for post-processing and conversion of trace data, is employed here to efficiently filter and organize the incoming streams into a unified format, enabling downstream analysis and visualization. This adaptation is critical for applications where latency and synchronization are paramount, such as in distributed sensor networks or multi-camera video analysis systems. Furthermore, the proposed approach leverages timestamp alignment and metadata encoding within each trace packet, ensuring that trace events from different hardware units are accurately correlated. This is particularly advantageous in environments where hardware components operate at different clock speeds or where data processing involves heterogeneous systems. The combination of this multi-node trace gathering with the LTTng and Babeltrace framework not only enhances scalability but also provides a robust foundation for real-time delay measurement and bottleneck identification across distributed systems. By extending the functionality of established tools, this method bridges the gap between traditional single-node tracing and

the growing demand for comprehensive, synchronized multi-node analysis. In Figure 4, we can see that next task is the delay calculation task, as in Figure 4 (4). The task is a fairly easy process: We temporarily create empty variables of timestamps and component names. When we receive initial trace data, we skip the calculation process and store first trace data in our empty timestamp and component name variable (they were empty at first, but now they contain data of first received trace data). After acquiring the second round of trace data, with a second timestamp and component name, we start the calculation of the time difference. The calculation is simply finding the difference between the first and second timestamp. We make sure that we are calculating the time difference of the timestamps with the same component name. After calculations, we dump the initially stored timestamp and component name, and instead we temporarily store in the buffer the second timestamp and component name. The first timestamp and component name are no longer needed. Now, when the third round of timestamps and component name arrives, we proceed with the same procedure of calculating the time difference between the third and the second timestamps. All of that is happening in a loop. In such a way, we can calculate the delay for each component and the time between components.

The data organization in Figure 4 (5) involves structuring calculated delay and sending it to the delay visualization module that is running in parallel. First, we are storing entry point and exit point timestamps for each component in the data structure. The data structure also stores the component name and hardware name. After all the necessary calculations and delay data processing, the delay calculation module sends the delay data over to the delay visualization script using a "pipe" multiprocessing connection.

3.3. Delay Visualization Module

The delay visualization module is designed to enhance the comprehension of delay measurement results by converting complex, raw data into intuitive visual representations. In dynamic environments where prompt decision-making is critical, this module provides real-time insights by visually representing time lags or latency in the sensor data analysis process. By making delays more accessible and understandable, the module enables users to quickly grasp the impact of delays on system behavior and performance, ultimately facilitating better human comprehension and more informed decision-making.

The operation of the delay visualization module, as shown in part of Figure 4b, begins with launching the server in Figure 4 (6), which establishes the necessary communication infrastructure for real-time data transfer. A WebSocket [34] connection is then established in Figure 4 (7) to ensure continuous and efficient communication between the delay calculation module and the visualization client. This real-time data pipeline allows for immediate transmission and updating of delay data without noticeable latency, providing an accurate and up-to-date reflection of the system's performance. Once the data are processed and organized by the delay calculation module, they are sent to the client (Figure 4 (8)) for visualization (Figure 4 (9)). The client, typically a web-based interface, then graphically represents the delay data, enabling users to intuitively analyze and interpret performance metrics. This process ensures that complex data are translated into actionable insights, offering a user-friendly interface that supports real-time monitoring and analysis.

The visualization module GUI is designed as a horizontal bar chart, where each bar represents a specific process within the sensor data analysis pipeline. The width of each bar corresponds to the duration of the process, determined by the starting and ending timestamps on the horizontal axis, which is represented in milliseconds. The GUI must update constantly as new delay calculation data arrive, ensuring that the visualization remains accurate and responsive. To achieve this, the module utilizes a combination of WebSockets and JavaScript, technologies that support the fast, real-time processing required

for handling large-scale data. WebSockets enable ongoing interactive communication between the client and server, while JavaScript facilitates the creation of dynamic and interactive visualizations.

The visualization module is tightly integrated with the delay calculations script. Upon launching, the server runs on a designated port (e.g., 3000), and a WebSocket connection is established using simple functions from the "socketio" library. The server then waits for structured trace data from the delay calculation module. Once the data are received, they are transmitted to the client, where the JavaScript-based GUI updates the visualization in real time. The bars on the interface represent individual processes in sensor data analysis, with their widths dynamically adjusted based on the current data. The time axis also updates to reflect the most recent timestamps, ensuring that the visualization remains synchronized with the ongoing data analysis.

3.4. Algorithms of Delay Measurement Tool

This subsection describes the delay measurement tool's algorithms. The algorithms focus on the three software modules: software tracing, delay calculation, and delay visualization modules. The provided algorithms represent a high-level operation order of each software module of the proposed software tool.

Algorithm 1 illustrates the refined implementation of the software tracing module within our delay measurement tool, designed to operate seamlessly in the target system based on the DeepStream pipeline [32]. The target application represents a simplified sensor data analysis process using synchronized video streams, which serves as a use case for this study. However, the tracing methodology can be generalized to other sensor data-processing applications as long as the source code is accessible. The tracing process begins in Software_Tracing() (line 1), where the Target_Application() function is called to initiate the sensor data analysis while enabling the tracing mechanism. The key functionality of the tracing module lies in the Record_Tracepoint() and Tracepoint() methods, which are designed to log component timestamps and send trace data to monitoring hardware with minimal system interference. Within Target_Application() (lines 5–15), the total number of frames to process is defined based on the input sensor data, controlling the execution loop. Each iteration (line 6) processes a frame by sequentially executing two main tasks (Application_Task_1() and Application_Task_2()), which represent abstract sensor analysis operations utilizing the DeepStream SDK API [32]. At the start and end of each task (lines 7–9 and 11–13), Record_Tracepoint() is invoked to log the component name (e.g., "start_application_task_1") and a precise hardware timestamp captured using get_current_timestamp(). These parameters are packaged into a tracepoint object, which is passed to the Tracepoint() method to generate a TCP_packet containing component names, timestamps, and hardware identifiers. The TCP_packet is then transmitted over ethernet to the monitoring system (lines 23–24) for delay measurement and further analysis. The modular design of the algorithm ensures that the tracing mechanism operates in parallel with the target application without interfering with its execution flow. Record_Tracepoint() encapsulates the tracepoint creation logic, while Tracepoint() prepares and transmits the trace data for monitoring. This modularity enhances clarity, minimizes overhead, and ensures efficient logging of runtime behavior. By analyzing the timestamps of multiple tracepoints, delays between various tasks or components within the target application can be accurately measured, providing insights into the system's performance and potential bottlenecks.

Algorithm 1. Software Tracing Module

Input: DA (Data_Array): Array of data for analysis in target software
TP (trace parameters): Component name and timestamp

Output: TCP_packet: Trace data sent to monitoring hardware (over ethernet)
RDA (result of data analysis): Output of target application (non-integral to the tracing module)

Method:

```
01: Software_Tracing(DA):
02:     RDA, TCP_packet ← Target_Application(DA);
03:     Return (RDA, TCP_packet);

04: Target_Application(DA):
05:     frame_count ← number_of_frames(DA);//Total frames in video file or stream
06:     For i ← 1 to frame_count do:
07:         Record_Tracepoint("start_application_task_1");
08:         Intermediate_Result ← Application_Task_1(DA[i]);//DeepStream SDK Task 1
09:         Record_Tracepoint("end_application_task_1");
10:
11:         Record_Tracepoint("start_application_task_2");
12:         RDA ← Application_Task_2(Intermediate_Result);//DeepStream SDK Task 2
13:         Record_Tracepoint("end_application_task_2");
14:     End For
15:     Return (RDA, TCP_packet);

16: Record_Tracepoint(component_name):
17:     timestamp ← get_current_timestamp();
18:     TP ← (component_name, timestamp);
19:     TCP_packet ← Tracepoint(TP);
20:     Return (TCP_packet);

21: Tracepoint(TP):
22:     component_name, tracepoint ← TP;
23:     TCP_packet ← (component_name, tracepoint, hardware_name);
24:     sendToMonitoringHW(TCP_packet, monitoringHW_IPaddress);
25:     Return (TCP_packet);
```

The computational complexity of the proposed algorithms was analyzed to evaluate their scalability and efficiency in real-time, multi-node environments. Algorithm 1, which pertains to the software tracing module, inserts tracepoints at critical locations within the target application to capture timestamps and component names for each data frame. Let n represent the number of tracepoints and t the number of iterations (i.e., in our case, this corresponds to the number of processed frames). The time complexity of this algorithm is $O(n \times t)$, as tracepoints are executed for each frame, and its space complexity is $O(n \times t)$, reflecting the storage requirements for trace data until transmission. This linear complexity ensures minimal computational overhead while maintaining efficiency for

real-time processing. The goal of Tracepoint() is to prepare and transmit data for delay calculation, as outlined in Algorithm 2.

Algorithm 2. Delay Calculation & Visualization Modules

Input: TCP_Packets: A dictionary containing trace data packets from multiple hardware nodes, keyed by
Hardware_Name. Example: TCP_Packets = {"ENDBOX1": TCP_Packet1, "ENDBOX2": TCP_Packet2, . . .}
Delay Data–results of delay calculation, that are used for visualization
Output: Status: The status of delay visualization.

Method:

```
Delay_Calculation(Hardware_Names):
    Previous_Component ← {};//Store previous components for each hardware node
    Previous_Timestamp ← {};//Store previous timestamps for each hardware node
    TCP_Packets ← {};//Initialize empty packet storage for all nodes

//Step 1: Collect packets using babeltrace message iterators
    For Hardware_Name in Hardware_Names:
        TCP_Packets[Hardware_Name] ← Receive_Packets(Hardware_Name);//Synchronous, sequential
message retrieval
    End For

//Step 2: Process packets for delay calculation
    For Hardware_Name in TCP_Packets:
        For Packet in TCP_Packets[Hardware_Name]:
            Component_Name, Timestamp ← Extract(Packet);
            If Previous_Component[Hardware_Name] == Component_Name:
                Delay ← Timestamp-Previous_Timestamp[Hardware_Name];
                Delay_Data←Hardware_Name,Component_Name,
Previous_Timestamp[Hardware_Name], Timestamp, Delay;
                Status ← Delay_Visualization(Delay_Data);
            End If

            Previous_Component[Hardware_Name] ← Component_Name;
            Previous_Timestamp[Hardware_Name] ← Timestamp;
        End For
    End For

    Return (Status);

Delay_Visualization(Delay_Data)://Visualizes delays using horizontal bars
    Hardware_Name, Component_Name, Previous_Timestamp, Timestamp, Delay
←
Extract(Delay_Data);
    Horizontal_Bar ← Draw_Bar(Hardware_Name, Component_Name);
    Status ← Horizontal_Bar.Latency.Update(Previous_Timestamp, Timestamp);

    Return (Status);
```

The delay calculation and visualization modules algorithm, outlined in Algorithm 2, describes the process of calculating and visualizing delays in a multi-node environment. The input to this algorithm, Hardware_Names, represents the list of hardware nodes being monitored, and TCP_Packets is the trace data collected from each node. The algorithm employs Babeltrace-based message iterators to retrieve trace packets in a synchronous and sequential manner for each node. At the beginning (lines 2–4), two empty dictionaries, Previous_Component and Previous_Timestamp, are initialized to store the last observed component name and its corresponding timestamp for each hardware node. Additionally, an empty dictionary TCP_Packets is prepared to collect trace data. In Step 1 (lines 6–9), the algorithm iterates through all Hardware_Names and uses the function Receive_Packets(Hardware_Name) to retrieve trace data. This function utilizes the Babeltrace message iterator, which efficiently processes trace events in a stream-like fashion. The iterator reads events from trace files, extracting component names and their corresponding timestamps with nanosecond precision. This synchronous retrieval ensures that trace packets for all hardware nodes are collected without loss or misalignment. Step 2 (lines 12–24) processes the retrieved packets to calculate delays. For each hardware node, the algorithm iterates through its trace packets (line 13). The function Extract(Packet) unpacks the trace data, extracting the Component_Name and Timestamp. At line 15, the algorithm checks if the current Component_Name matches the previous component stored for the corresponding hardware node. If it does, the delay is calculated (line 16) by subtracting the previously stored timestamp (Previous_Timestamp) from the current timestamp. This calculated delay, along with metadata such as the hardware name, component name, previous timestamp, and current timestamp, is bundled into Delay_Data (line 17). The packaged delay data is then sent to the delay visualization module for graphical representation (line 18). The delay visualization module (lines 27–31) extracts the delay data and updates a horizontal bar graph using the Draw_Bar function, where each bar corresponds to a specific hardware node and its components. The function Latency.Update dynamically updates the visualization, incorporating the previous and current timestamps to accurately reflect the measured delays. By combining Babeltrace-based message iterators for trace retrieval and efficient delay processing, the algorithm ensures a clear, synchronized analysis of delays across multiple nodes. This method provides system administrators with an intuitive and continuous representation of performance bottlenecks, enabling real-time identification and resolution of delays. For Algorithm 2, which pertains to the delay calculation and visualization module, the key operations include receiving trace data from multiple hardware nodes, extracting timestamps, calculating delays, and updating the visualization module. Let h represent the number of hardware nodes, n the number of tracepoints, and t the number of iterations or time steps (i.e., in our case, this corresponds to the number of processed frames). The time complexity of this algorithm is $O(h \times n \times t)$, as delays are calculated for each node and tracepoint over time, while the space complexity is $O(h \times n)$, corresponding to the storage of trace data and calculated delays. This linear scaling across nodes and tracepoints ensures the modular design of the visualization module can handle real-time scenarios efficiently. These analyses demonstrate that the algorithms are computationally efficient and scalable, making them well suited for distributed systems requiring synchronized monitoring and delay measurement. This comprehensive approach not only enhances the visibility of system performance but also facilitates proactive management of system delays.

3.5. Temporal Synchronization

Hardware clock synchronization is critical in distributed computing systems to ensure accurate and consistent timing across devices, which is essential for the proposed delay

measurement tool. The tool relies on synchronized hardware clock timestamps to calculate delays accurately. Without synchronization, discrepancies between device clocks can lead to errors in delay measurements, misrepresenting the sequence or timing of events. For instance, if one clock is ahead, events may appear to have occurred earlier than they actually did, while a lagging clock may record events as delayed or miss them altogether.

Precision Time Protocol (PTP) [35] is widely used to achieve high-precision clock synchronization. PTP synchronizes device clocks with sub-microsecond accuracy by designating a master clock as the network's time reference and aligning slave clocks through timestamp exchanges and correction mechanisms. This process compensates for network latency and clock drift, ensuring consistent timestamps across devices. Implementing PTP requires specialized hardware, including PTP-capable network interface cards and switches, as well as proper software configuration. By leveraging PTP, the delay measurement tool ensures reliable synchronization, enabling precise delay analysis and enhancing the performance of distributed systems.

Figure 5 depicts the architecture of two data concentrator units (DCUs) within a synchronized video frame data-processing system. The use of dual DCUs—(a) DCU1 and (b) DCU2, shown in Figure 5—ensures high availability (HA) [36], a crucial feature in autonomous vehicle systems. This redundancy allows the system to maintain continuous operation by enabling one DCU to take over if the other fails, ensuring safety and reliability. Each DCU interfaces with a base board that manages data flow from multiple synchronized cameras. The cameras are arranged to capture video data from different angles, providing comprehensive environmental coverage. The base boards ensure time alignment of video frames, enabling accurate sensor fusion and data analysis, which is essential for real-time decision-making.

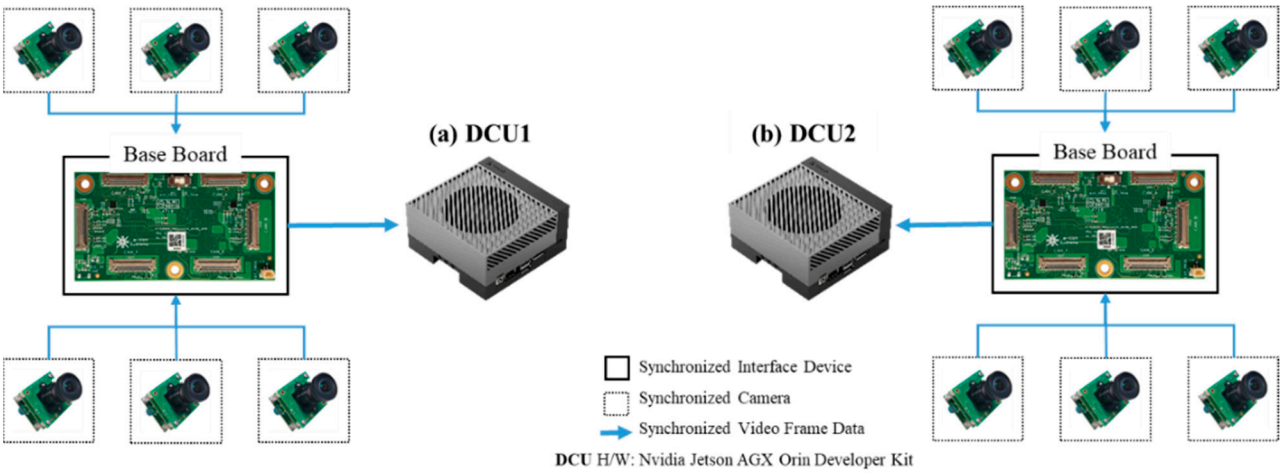

Figure 5. Design of synchronized cameras and DCU connection diagram for high availability (HA).

In Figure 5, blue arrows trace the synchronized video frame data flow from the cameras to the DCUs, emphasizing the importance of precise timing in multi-camera setups. The Jetson AGX Orin Developer Kit, shown in both DCUs, provides high-performance computing with low power consumption, suitable for autonomous vehicle solutions. The e-con Systems multi-camera setup (e.g., e-CAM20_CUOAGX) is connected via base boards to the DCUs, with external trigger support for synchronization. Synchronization across DCUs is achieved using the phc2sys program [37], which aligns the system clock with the PTP hardware clock (PHC) on the NIC (network interface card). By combining PTP-based clock synchronization and e-con Systems' multi-camera synchronization, the architecture achieves precise temporal alignment, enhancing object detection and overall performance for autonomous vehicle applications.

Precision Time Protocol (PTP) synchronization was configured and validated using a Supermicro SYS-210P server configured as the grandmaster clock (i.e., monitoring system) and two NVIDIA Jetson AGX Orin devices as slave clocks (i.e., target systems). All devices were interconnected via a PTP-capable ethernet switch to ensure accurate communication for time synchronization. The SYS-210P was configured with the ptp4l tool operating in grandmaster mode, which allowed the local clock (3cecef.fffe.bbd305) to be selected as the best master clock. This configuration enabled the grandmaster to broadcast synchronization messages, such as Sync and Follow_Up packets, to all connected devices. The successful initialization and transition of the SYS-210P to the grandmaster role are depicted in Figure 6, confirming its capability to maintain a centralized timing reference for the system.

```
netai@edgebox1:~$ sudo ptp4l -i eno2 -m --masterOnly 1
option masterOnly is deprecated, please use serverOnly instead
ptp4l[523038.087]: selected /dev/ptp1 as PTP clock
ptp4l[523038.088]: port 1 (eno2): INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[523038.088]: port 0 (/var/run/ptp4l): INITIALIZING to LISTENING on INIT_COMPLET
E
ptp4l[523038.088]: port 0 (/var/run/ptp4lro): INITIALIZING to LISTENING on INIT_COMPL
ETE
ptp4l[523046.060]: port 1 (eno2): LISTENING to MASTER on ANNOUNCE_RECEIPT_TIMEOUT_EXP
IRES
ptp4l[523046.060]: selected local clock 3cecef.fffe.bbd305 as best master
ptp4l[523046.060]: port 1 (eno2): assuming the grand master role
```

Figure 6. PTP master clock initialization on the monitoring system.

The NVIDIA Jetson AGX Orin devices (NVIDIA Corporation, Santa Clara, CA, USA) were configured in slave-only mode using the ptp4l tool to synchronize their hardware clocks (/dev/ptp0) with the grandmaster clock. Logs from one of the slave devices, shown in Figure 7, demonstrate the detection and selection of the grandmaster clock, with the port transitioning to the SLAVE state. The master offset values steadily converged to small values (e.g., ~100 ns), indicating precise alignment with the grandmaster. Additionally, consistent path delays (e.g., ~5900–6500 ns) validated stable network conditions, essential for reliable round-trip time measurements. Dynamic frequency adjustments were observed as the slave device continuously corrected its clock drift, ensuring sub-microsecond accuracy. Similar configurations and validation steps were performed on the second NVIDIA Jetson AGX Orin device, with comparable results confirming its synchronization with the master clock.

To further refine synchronization, the phc2sys tool was employed to align the system clock (CLOCK_REALTIME) with the PTP hardware clock on the NVIDIA Jetson device. The results, captured in Figure 8, reveal offsets between the system clock and hardware clock remaining consistently within a narrow range (e.g., −209 to 322 ns). Dynamic frequency corrections minimized drift, while stable path delays (~2600 ns) confirmed reliable communication between the system and hardware clocks. Overall, the combination of PTP clock synchronization and e-con Systems' multi-camera synchronization ensures temporal alignment that enhances object detection for AV.

```
netal@endbox2:~$ sudo ptp4l -i eth0 -s -m -f /etc/ptp4l.conf
ptp4l[81467.674]: selected /dev/ptp0 as PTP clock
ptp4l[81467.680]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[81467.680]: port 0: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[81468.130]: port 1: new foreign master 3cecef.fffe.bbd305-1
ptp4l[81472.069]: selected best master clock 3cecef.fffe.bbd305
ptp4l[81472.069]: port 1: LISTENING to UNCALIBRATED on RS_SLAVE
ptp4l[81474.467]: master offset -10703241531 s0 freq   -5263 path delay      6760
ptp4l[81475.472]: master offset -10703246048 s1 freq   -9780 path delay      7385
ptp4l[81476.519]: master offset       4757 s2 freq   -5023 path delay      7385
ptp4l[81476.519]: port 1: UNCALIBRATED to SLAVE on MASTER_CLOCK_SELECTED
ptp4l[81477.619]: master offset       5360 s2 freq   -2993 path delay      7385
ptp4l[81478.720]: master offset       4408 s2 freq   -2337 path delay      7072
ptp4l[81479.820]: master offset       1969 s2 freq   -3453 path delay      7072
ptp4l[81480.919]: master offset       1107 s2 freq   -3725 path delay      6598
ptp4l[81482.019]: master offset         90 s2 freq   -4409 path delay      6511
ptp4l[81483.119]: master offset        151 s2 freq   -4321 path delay      6511
ptp4l[81484.220]: master offset        230 s2 freq   -4197 path delay      6272
ptp4l[81485.320]: master offset         37 s2 freq   -4321 path delay      6319
ptp4l[81486.420]: master offset       -206 s2 freq   -4553 path delay      6319
ptp4l[81487.520]: master offset       -489 s2 freq   -4898 path delay      6165
ptp4l[81488.620]: master offset       -126 s2 freq   -4682 path delay      6047
ptp4l[81489.720]: master offset       -513 s2 freq   -5106 path delay      6047
ptp4l[81490.820]: master offset        250 s2 freq   -4497 path delay      6047
ptp4l[81491.921]: master offset        832 s2 freq   -3840 path delay      5926
ptp4l[81493.020]: master offset       -478 s2 freq   -4901 path delay      5926
ptp4l[81494.121]: master offset        120 s2 freq   -4446 path delay      5926
ptp4l[81495.032]: master offset        144 s2 freq   -4386 path delay      5926
ptp4l[81496.112]: master offset       -237 s2 freq   -4724 path delay      5926
ptp4l[81497.213]: master offset       -515 s2 freq   -5073 path delay      5852
ptp4l[81498.312]: master offset       -279 s2 freq   -4991 path delay      5852
ptp4l[81499.412]: master offset        213 s2 freq   -4583 path delay      5876
ptp4l[81500.513]: master offset        511 s2 freq   -4221 path delay      5876
ptp4l[81501.613]: master offset        275 s2 freq   -4304 path delay      5876
ptp4l[81502.713]: master offset       -276 s2 freq   -4772 path delay      5849
ptp4l[81503.813]: master offset       -474 s2 freq   -5053 path delay      5835
ptp4l[81504.913]: master offset        580 s2 freq   -4141 path delay      5789
ptp4l[81506.013]: master offset        171 s2 freq   -4376 path delay      5795
ptp4l[81507.113]: master offset       -251 s2 freq   -4747 path delay      5802
ptp4l[81508.213]: master offset         23 s2 freq   -4548 path delay      5802
ptp4l[81509.313]: master offset       -522 s2 freq   -5087 path delay      5883
ptp4l[81510.413]: master offset        781 s2 freq   -3940 path delay      5832
ptp4l[81511.513]: master offset       -228 s2 freq   -4715 path delay      5951
ptp4l[81512.613]: master offset       -577 s2 freq   -5132 path delay      5986
ptp4l[81513.714]: master offset       -204 s2 freq   -4932 path delay      5986
ptp4l[81514.814]: master offset        675 s2 freq   -4115 path delay      5847
ptp4l[81515.914]: master offset       -350 s2 freq   -4937 path delay      5847
ptp4l[81517.014]: master offset        605 s2 freq   -4087 path delay      5826
ptp4l[81518.114]: master offset        324 s2 freq   -4187 path delay      5826
ptp4l[81519.214]: master offset       -557 s2 freq   -4970 path delay      5826
ptp4l[81520.314]: master offset        279 s2 freq   -4301 path delay      5882
ptp4l[81521.414]: master offset       -333 s2 freq   -4830 path delay      5882
ptp4l[81522.514]: master offset       -119 s2 freq   -4716 path delay      5882
```

Figure 7. PTP slave clock initialization on the target system.

```
phc2sys[83326.512]: CLOCK_REALTIME phc offset       322 s2 freq   -5212 delay   2624
phc2sys[83327.512]: CLOCK_REALTIME phc offset       292 s2 freq   -5145 delay   2688
phc2sys[83328.512]: CLOCK_REALTIME phc offset      -421 s2 freq   -5771 delay   2688
phc2sys[83329.513]: CLOCK_REALTIME phc offset      -244 s2 freq   -5720 delay   2720
phc2sys[83330.513]: CLOCK_REALTIME phc offset       363 s2 freq   -5186 delay   2720
phc2sys[83331.514]: CLOCK_REALTIME phc offset      -148 s2 freq   -5588 delay   2688
phc2sys[83332.514]: CLOCK_REALTIME phc offset       258 s2 freq   -5227 delay   2656
phc2sys[83333.515]: CLOCK_REALTIME phc offset      -393 s2 freq   -5800 delay   2720
phc2sys[83334.515]: CLOCK_REALTIME phc offset       407 s2 freq   -5118 delay   2720
phc2sys[83335.516]: CLOCK_REALTIME phc offset      -145 s2 freq   -5548 delay   2720
phc2sys[83336.516]: CLOCK_REALTIME phc offset       290 s2 freq   -5157 delay   2688
phc2sys[83337.517]: CLOCK_REALTIME phc offset      -451 s2 freq   -5811 delay   2592
phc2sys[83338.517]: CLOCK_REALTIME phc offset       238 s2 freq   -5257 delay   2560
phc2sys[83339.518]: CLOCK_REALTIME phc offset      -459 s2 freq   -5882 delay   2592
phc2sys[83340.518]: CLOCK_REALTIME phc offset       -90 s2 freq   -5651 delay   2592
phc2sys[83341.519]: CLOCK_REALTIME phc offset       560 s2 freq   -5028 delay   2592
phc2sys[83342.519]: CLOCK_REALTIME phc offset       259 s2 freq   -5161 delay   2656
phc2sys[83343.519]: CLOCK_REALTIME phc offset       166 s2 freq   -5176 delay   2592
phc2sys[83344.520]: CLOCK_REALTIME phc offset      -299 s2 freq   -5592 delay   2592
phc2sys[83345.520]: CLOCK_REALTIME phc offset      -825 s2 freq   -6207 delay   2560
phc2sys[83346.521]: CLOCK_REALTIME phc offset        44 s2 freq   -5586 delay   2624
phc2sys[83347.521]: CLOCK_REALTIME phc offset        85 s2 freq   -5532 delay   2688
phc2sys[83348.522]: CLOCK_REALTIME phc offset       715 s2 freq   -4876 delay   2496
phc2sys[83349.522]: CLOCK_REALTIME phc offset        13 s2 freq   -5364 delay   2656
```

Figure 8. Synchronization between the hardware clock and the system clock on the target system.

4. Experiment and Evaluation

This section details the experimental environments, evaluation methods, and results. The primary aim of the experiment is to demonstrate the capability of the delay measurement tool in monitoring multiple data concentrator units (DCUs) simultaneously. The first subsection provides an in-depth overview of the experimental environments, including the hardware and software configurations. The second subsection describes the experiments data and evaluation measure for the evaluation methods. The final subsection presents a detailed discussion of the experiment's results, highlighting the tool's performance and effectiveness in real-time monitoring and data analysis across multiple DCUs.

4.1. Experiment Environments

This subsection details the experimental setup, including the hardware and software environments used to evaluate the delay measurement tool's performance. The primary objective of the experiment is to demonstrate the tool's ability to simultaneously monitor multiple DCUs, a task requiring substantial computational resources. To accurately test and validate the delay measurement tool, the tool's functionality involves real-time monitoring and analysis of sensor data, which are inherently computationally intensive processes.

The experiment setup consists of a monitoring system and a target system, with the Supermicro SYS-210P and Nvidia Jetson AGX Orin used, respectively. The Supermicro SYS-210P offers powerful dual-processor performance and extensive memory capacity in a compact 2U design, making it ideal for space-constrained data centers that require high efficiency without sacrificing computational capabilities. Meanwhile, the Nvidia Jetson AGX Orin is a compact device that is designed for edge computing and AI inferencing applications, offering compact yet powerful processing capabilities. It is suited very well as the target system that performs sensor data analysis. The specific hardware configurations are detailed in Table 2, showcasing the differences between the monitoring and target systems used in the experiment.

Table 2. Hardware specification for experimental environments.

	Monitoring System	**Target System**
Hardware model	Supermicro SYS-210P (server-grade system)	Nvidia Jetson AGX Orin
Processor	Intel Xeon Silver 4310, 12-Dual core, 2.1 GHz	12-core Arm® Cortex®-A78AE v8.2 64-bit CPU 3 MB L2 + 6 MB L3, 2.2 GHz
CPU cores	12-Dual core	12-core
CPU architecture	x86-64 Architecture	ARMv8-A architecture
GPU model	Nvidia Tesla T4	2048-Core NVIDIA Ampere GPU 64 Tensor Cores, 1.3 GHz
Memory (RAM)	32 GB DDR4 memory	32 GB 256-bit LPDDR5
Operating system	Linux Ubuntu 20.04	Linux Ubuntu 20.04

Software components are another crucial part in our environment. In our experiment, we used a high-performance data pipeline to perform the object-detection analysis over multiple synchronized video streams, and those analyses would be the target our proposed delay measurement tool to monitor. The proposed tool itself has been built using multiple software components. Figure 9 shows the software configuration in detail, which presents the configurations of the modified open-source tools (i.e., LTTng [19] and Babeltrace2 [33]) used to build the delay measurement tool. The software tracing module in Figure 9a employs modified LTTng (Linux Trace Toolkit: next generation) as its core component, offering robust real-time tracing capabilities for analyzing sensor data. Software tracing module analyzes the execution of data analysis. In our case, for testing and experiments, we used Nvidia DeepStream to perform object-detection analysis. The collected trace data are transmitted to the delay calculation module, as shown in Figure 9b, managed by lttng-relayd, which ensures efficient data transmission and storage across the network. Following this, Babeltrace is used to process and convert the trace data, extracting essential performance metrics such as timestamps, which are then used for delay calculations. The results of these calculations are transmitted via WebSockets, enabling real-time data flow to the delay visualization module, as shown in Figure 9c. This visualization is rendered using

JavaScript, providing an interactive and comprehensible visual representation of the delays, allowing users to easily interpret the system's performance metrics. The entire system is deployed on an embedded platform powered by NVIDIA's JetPack 5.1 and runs on Ubuntu 20.04 as the operating system. The platform is supported by the Linux kernel version 5.10, CUDA 11.4 for GPU acceleration, and cuDNN 8.6.0 for deep learning optimizations. This configuration ensures the system's high performance and reliability, making it a powerful tool for measuring and visualizing system delays in performance-critical applications.

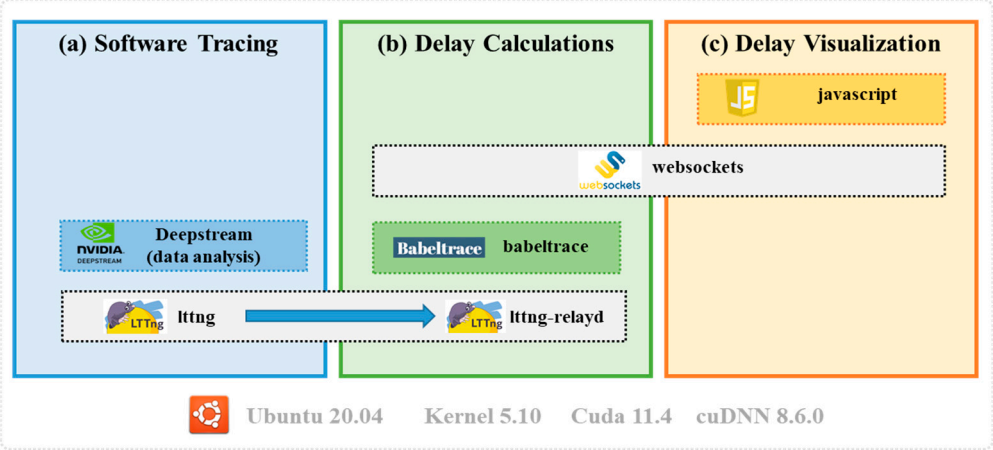

Figure 9. Leveraging open-source software for the delay measurement tool.

Figure 9 provides a comprehensive view of the prototype delay measurement environment, showcasing both the hardware and the software components used in the testing setup. The environment consists of several key elements, each labeled in the image. On the left, the monitoring system (a) is highlighted, which is responsible for collecting and processing data from the target systems. Adjacent to it, the target systems (b) are shown, consisting of devices where the delay measurement tool is implemented and tested. In front of the setup, the synchronized cameras (c) capture live video streams, simulating real-world data input for the system. The setup also includes a large display showing the result of object detection (d), demonstrating the tool's application in analyzing visual data. Additionally, a screen displaying the delay visualization (e) is visible, where processed delay data are graphically represented, allowing for real-time monitoring and analysis. This integrated environment effectively combines hardware and software to evaluate the performance of the delay measurement tool in a realistic and controlled setting.

Figure 10 illustrates the research and development environment utilized in this paper. The environment is designed to test and evaluate the delay measurement tool's capabilities. Figure 10a showcases the target systems that form the core of our hardware setup, vital for executing the sensor data analysis. Adjacent to this, Figure 10b depicts the synchronized cameras that are integral for capturing simultaneous video feeds, crucial for the accurate analysis of object detection algorithms. Figure 10c presents the PreScan Virtual Simulator, a key component that provides a simulated environment, aiding in the calibration and testing of our delay measurement tool and sensor data analysis under controlled conditions. The object detection results are displayed on the monitor in Figure 10d, where result from multiple cameras reveal the outcomes of processing by data pipeline. Finally, Figure 10e shows the delay visualization tool, a setup that graphically represents the processing delays, providing a visual assessment and facilitating further optimization of the system's performance. This integrated environment not only supports the development of proposed delay measurement tool but also ensures comprehensive testing across various scenarios.

Figure 10. Prototype of the delay measurement environment with H/W and S/W.

4.2. Experiment Configuration and Evaluation Measure

In our experiments, we utilized two distinct types of video stream data to evaluate the performance of the proposed tool. The first data type comprises multiple live, synchronized video streams captured using synchronized cameras. This setup is designed to closely mimic real-time applications where low latency and precise synchronization are critical. The second dataset consists of the pre-recorded video streams stored in a target system. While our primary focus is on assessing the tool's performance with live streaming data due to their relevance in real-time scenarios, we also conducted tests using stored video streams. The experiment scenarios tested a range of streams, from one to six, for each data type and frame size, allowing for a comprehensive evaluation of how the system handles varying workloads. This systematic approach provides insights into the tool's ability to manage different data rates and stream counts, highlighting its robustness and adaptability in diverse operating conditions. The structured comparison between live video stream data and stored video file data also offers a valuable perspective on the tool's performance across different data sources, contributing to a thorough understanding of its capabilities. This approach allows us to provide a comprehensive evaluation of the tool's versatility and effectiveness across different types of data. By doing so, we aim to offer insights into how the tool performs under varying conditions, thereby highlighting its robustness and adaptability.

We used the MOT17 dataset [38] and the MOT20-02 sequence [39] for testing, chosen for their established use in object tracking and diverse urban scene complexity. Both datasets provide high-resolution (1920 $\times$ 1080) images, suitable for evaluating system performance under real-world conditions. Using FFmpeg [40], the images were converted into 30 FPS videos, with sequences concatenated into continuous clips to ensure seamless transitions and consistent quality. Only unique 1920 $\times$ 1080 frames were included, avoiding duplication. To simulate higher throughput, the videos were upsampled to 60 FPS via frame interpolation, maintaining consistent durations (e.g., 30 s for both 30 and 60 FPS) and normalized bitrates. This allowed for a fair comparison of system performance under different frame rates, focusing on processing efficiency. The rationale for interpolating 30 FPS videos to 60 FPS was to align testing conditions for delay measurement between video files and synchronized streams. Our synchronized cameras operate exclusively at 30 and 60 FPS, ensuring consistency between video data and real-time processing. While this study utilized JPEG images and their derived video streams to ensure clean and standardized inputs, the incorporation of advanced video compression techniques presents

an opportunity for future work. Methods such as those described by Wiseman Y. [41], which prioritize machine vision requirements over human perception, could enable seamless integration of real-time video streams while maintaining computational efficiency. Future developments could address challenges such as compression latency and adaptation to synchronized multi-sensor setups.

The Table 3 outlines the experiment's data types, specifying parameters for video stream and video file data, both maintaining a resolution of 1920 × 1080 pixels. Frame rates of 30 and 60 FPS were tested, each processed for 250 s across one to six streams. The frames from cameras were YUV422 [42] 16-bit image format with UYVY [43] ordering, making a single frame size approximating 4.15 MB. Consequently, 30 FPS and 60 FPS streams amounted to 124.5 MB and 249 MB, respectively. For video files, the 30 FPS video was encoded at a consistent bitrate of 3986 kbps to ensure uniform data rates with a 60 FPS counterpart, isolating the effect of frame rate on performance. Additionally, 60 FPS files were evaluated at two bitrates (3986 kbps and 4971 kbps) to study the impact of data rates on system delay. Using the NVIDIA DeepStream SDK, a simple data pipeline with the YOLOv3 model was implemented for object detection on 80 COCO classes. The pipeline output was configured as a fakesink to focus on runtime monitoring and delay measurement. While object detection was verified visually, the primary goal remained runtime performance evaluation rather than accuracy testing.

Table 3. Data types used in the experiment.

Data Type	Resolution	Frames	Bit Rate (kbps) /Size (MB)	Duration (Seconds)	Number of Streams
Video stream data	1920 × 1080	30 60	124.5 249	250	From 1 to 6
Video file data	1920 × 1080	30 60 60	3982 (124.8) 3982 (124.8) 4971 (155.7)	250	From 1 to 6

To measure the overall delay in our system, we employed a straightforward yet effective method, as represented by the formula shown below:

$$S = \sum_{i=1}^{n} (T_i - T_{i-1}) \tag{1}$$

where the S denotes the total accumulated delay. The delay for each interval is calculated by subtracting the preceding timestamp (T_{i-1}) from the current timestamp (T_i). The summation is then performed over all time intervals from i = 1 to n, where n represents the total number of timestamps considered in the measurement. This approach allows us to accurately quantify the cumulative delay over a sequence of events or data points, providing a clear metric for assessing the temporal performance of the system. The simplicity of this method ensures that it can be easily applied across various scenarios, making it a versatile tool for delay measurement. However, to effectively present the results of our delay measurement, we utilized Formula (2).

$$A = \frac{S}{n} = \frac{\sum_{i=1}^{n} (T_i - T_{i-1})}{n} \tag{2}$$

where A represents the arithmetic mean of the accumulated delay per frame. Specifically, the mean delay is calculated by dividing the total sum of delays, denoted as S from Formula (1). By computing the arithmetic mean, we are able to concisely summarize

the average delay encountered during the sensor data analysis process. This metric is particularly useful, as it provides a clear measure of the overall delay performance, making it easier to assess the system's efficiency in handling time-sensitive data. The use of the arithmetic mean allows us to convey the typical delay experienced per frame, offering valuable insight into the temporal characteristics of the system. The experimental setup emulated the challenges of autonomous vehicle (AV) navigation in urban environments, such as high traffic density, dynamic obstacles, and frequent interactions with road users. Frame rates of 30 FPS and 60 FPS were chosen to reflect industry standards: 30 FPS for general perception tasks in moderate traffic and 60 FPS for high-speed object detection and decision-making in dynamic scenarios. Synchronized cameras ensured temporal coherence, aligning data streams from sensors like cameras, LiDAR, and radar for accurate sensor fusion and trajectory planning. Misaligned or delayed streams can lead to errors in object detection and obstacle avoidance, especially in unpredictable scenarios like pedestrian crossings or sudden lane changes. This study evaluated the delay measurement tool's ability to analyze latency and temporal alignment in synchronized video pipelines for real-time AV applications. While focused on video streams, the methodology is adaptable to other sensor modalities, like LiDAR and radar, which face challenges such as asynchronous data acquisition. Future work will extend the tool's capabilities to evaluate multi-sensor AV systems comprehensively.

4.3. Experimental Results

In this section, we present the results of our delay measurement tool designed to assess the latency during sensor data analysis, specifically focused on synchronized video and camera streams. The tool was developed to measure the delay introduced during the processing of multiple video streams under varying frame rates and bitrates, offering insights into how different factors impact system latency. In the first experiment, we evaluated the performance of the DCU by analyzing delays when processing video files of varying frame rates and bitrates (i.e., DCU load test). In the second phase, we extended the experiments to include synchronized video streams at frame rates of 30 FPS and 60 FPS, allowing us to compare the delay measurements between video streams and pre-recorded video files. The number of streams was progressively increased from one to six, providing a comprehensive view of how the system scales with additional computational load. Delays were calculated as the arithmetic mean per frame using Formula (2), as referenced earlier in the paper, providing a clear metric for understanding how the DCU handles increasing workloads under different frame rate and bitrate conditions.

The experimental results are detailed across three tables: Figure 11 and Table 4 focus on the delay measurements exclusively for video file data with respect to DCU load test, while Tables 5 and 6 compare the latency of both video stream and video file data at different frame rates. These tables illustrate the average delay per frame across various experimental setups, highlighting key trends and factors that influence system performance. This analysis is essential for evaluating the effectiveness of the delay measurement tool in real-time applications, where minimizing latency is critical to maintaining high responsiveness and system efficiency.

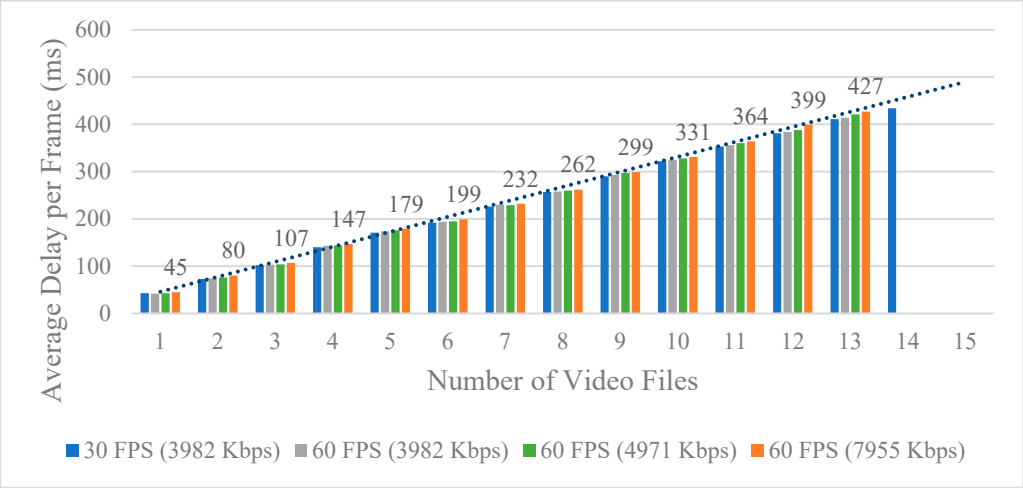

Figure 11. DCU load test by increasing the number of video files.

Table 4. Average delay (ms) per frame analysis for video file data.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Video file data 30 FPS (3982 kbps)	43	73	102	140	171	192	226	257	290	322	353	381	411	434	0
Video file data 60 FPS (3982 kbps)	42	74	103	143	174	194	230	258	293	325	356	384	414	0	0
Video file data 60 FPS (4971 kbps)	43	76	104	144	176	195	229	260	297	328	360	388	421	0	0
Video file data 60 FPS (7955 kbps)	45	80	107	147	179	199	232	262	299	331	364	399	427	0	0

Table 4 shows the average delay per frame analysis across multiple video file data conditions. The table distinctly categorizes video file data into two types based on frame rates of 30 FPS and 60 FPS but with varying bitrates: 3982 kbps, 4971 kbps, and a higher bitrate of 7955 kbps. This differentiation allows for a detailed assessment of how each variable, frame rate and bitrate, affects the latency performance of the DCU in processing video file data. In Table 4, DCU load tests show that it can process 14 and 13 video files simultaneously at 30 FPS and 60 FPS, respectively. For example, the video file data in the experiment may have had the same bit rate but a different frame rate. The data illustrated in Table 4 show a consistent pattern of increased delay with the augmentation of both the frame rate and the bitrate. For instance, video file data at 30 FPS and 3982 kbps start at an average delay of 43 ms for one video and escalate to 322 ms for ten videos. Similarly, at 60 FPS with the same bitrate, the delay commences at 42 ms, increasing to 325 ms, demonstrating how higher frame rates intensify the processing delay even under constant bitrate conditions. Notably, the impact of an increased bitrate at 60 FPS manifests in higher initial and final delays, starting at 43 ms and peaking at 331 ms for 7955 kbps, underscoring the additional processing overhead introduced by higher bitrates. The total average delays for each setting are calculated to provide a comparative insight, with the 30 FPS configuration at 3982 kbps exhibiting the least average delay, emphasizing the trade-offs involved when escalating frame rates and bitrates. This systematic analysis reveals that while increasing frame rates and bitrates can potentially improve video quality and detail, they correspondingly impose greater burdens on system latency, thus necessitating a balanced approach in real-time applications where low latency is crucial.

Table 5. Average delay per frame for the 30 FPS case.

# of Videos	1	2	3	4	5	6	Total Avg.
Video file data (30 FPS–3982 kbps)	43 ms	73 ms	102 ms	140 ms	171 ms	192 ms	120.17 ms
Video stream data (30 FPS)	41 ms	76 ms	107 ms	145 ms	174 ms	196 ms	123.16 ms

Table 5 presents a comparative analysis of average delay per frame for both video file data and video stream data at 30 frames per second. The data are arranged to display the incremental increase in delay as the number of video streams ranges from one to six. For both types of data—video file and video stream—the table shows that as the number of streams increases, there is a corresponding increase in delay, which is expected due to higher data-processing demands. Specifically, the video file data start with a delay of 43 ms for a single video and progress up to 192 ms for six videos, culminating in a total average delay of 120.17 ms. Similarly, video stream data begin at 41 ms and increase to 196 ms, with a total average of 123.83 ms. This slight difference in delay between file and stream data may be attributed to the real-time processing demands placed on the system by streaming data, which typically require more immediate decoding and rendering compared to file-based processing. The consistency in incremental delay across both datasets reaffirms the impact of increasing workload on system performance, emphasizing the need for efficient data-handling mechanisms in high-throughput video-processing environments. The total average delays give a quantitative baseline for evaluating the DCU's performance under varying operational stresses, providing insights critical for system optimization in real-time applications.

Table 6. Average delay per frame for the 60 fps case.

# of Videos	1	2	3	4	5	6	Total Avg.
Video file data (60 FPS–3982 kbps)	42 ms	74 ms	103 ms	143 ms	174 ms	194 ms	121.67 ms
Video file data (60 FPS–4971 kbps)	43 ms	76 ms	104 ms	144 ms	176 ms	195 ms	123 ms
Video file data (60 FPS–7955 kbps)	45 ms	80 ms	107 ms	147 ms	179 ms	199 ms	126.12 ms
Video stream data (60 FPS)	52 ms	82 ms	110 ms	149 ms	191 ms	206 ms	131.67 ms

Table 6 delivers an insightful exploration of average delays per frame for both video file data and video stream data at a 60 FPS rate, highlighting the impact of different bit rates on the system's latency. Presented across three distinct conditions for video file data with bit rates of 3982 kbps, 4971 kbps, and 7955 kbps, and a single consistent bitrate for video stream data, the table elucidates how increased frame rates and bit rates escalate processing demands within the DCU. With the number of video streams extending from one to six, an ascending trend in delays is observed across all settings. This rise in delay accentuates how the DCU copes with augmented processing loads, with the maximum delay observed at the highest bitrate of 7955 kbps for video file data, reaching up to 199 ms for six streams, illustrating a pronounced influence of bitrate on delay. In contrast, video stream data demonstrate a steady increase in delay, peaking at 206 ms for six streams, albeit less pronounced than the highest bitrate video file data condition. The total average delay

serves as a crucial metric, delineating the DCU's operational efficiency under varying data conditions. The 7955 kbps video file data condition records the highest total average delay at 126.12 ms, clearly indicating that higher bitrates significantly elevate latency. Conversely, video stream data, processed under a uniform bitrate, show a marginally higher total average delay of 131.67 ms, reflecting its consistent operational strain under continual data streaming. This analysis is vital for fine-tuning the DCU's performance in scenarios where low latency is paramount.

Figure 12 presents a comprehensive visualization of the average delay per frame, as influenced by variations in frame rate, bitrate, and the number of video streams. This bar chart clearly delineates the effects of different configurations across six incremental video stream counts, ranging from one to six streams. The colors differentiate the conditions: blue and green for video streaming data at 30 and 60 FPS, respectively, and shades of orange to yellow representing video file data across three different bitrates at 60 FPS. The graph elucidates a trend where delays generally escalate with the increase in the number of video streams, highlighting the system's increasing load. Notably, the transition from video file data at lower bitrates to higher reveals a progressive increase in delay, underscoring the impact of bitrate on processing times. This visual comparison aids in understanding how each variable frame rate, bitrate, and number of streams affects the performance of the delay measurement tool, providing critical insights for optimizing real-time video-processing applications where minimizing delay is crucial.

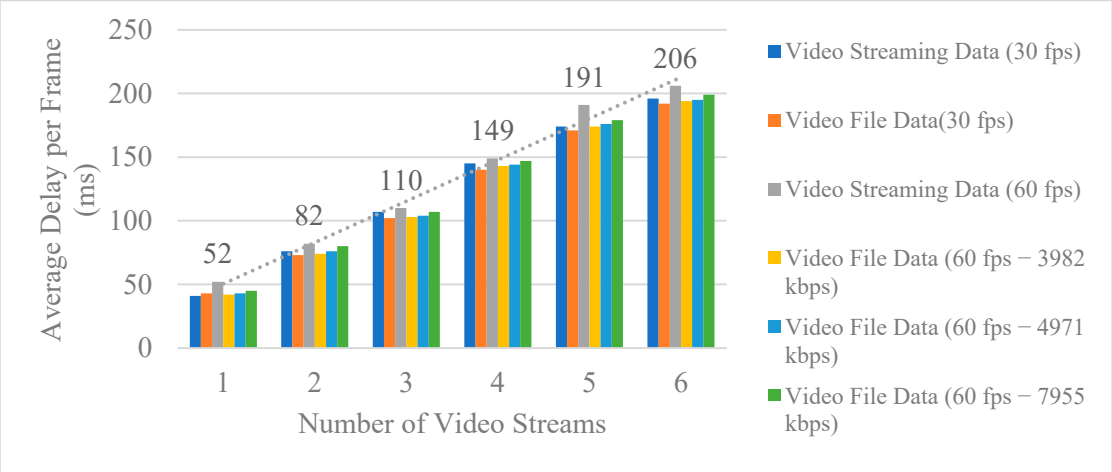

Figure 12. Comparing the results of delay measurement times.

4.4. Visualization of Delay Measurement Use Cases

The delay visualization module is an important component of our delay measurement tool, which designed to provide an intuitive and real-time representation of the system's delay performance. The purpose of this module is to offer users a clear and accessible way to understand the timing and latency characteristics of their system, enabling them to identify performance bottlenecks and optimize configurations effectively. In this paper, we presented two visualization versions: inner data pipeline task latency visualization and full data pipeline analysis visualization. The first visualization version shows the latency of separate tasks performed by the data pipeline during a single iteration (in this case, the batch is set at a single stream frame). For this specific version of visualization, we built an Nvidia data pipeline consisting of five Gstreamer plugins [44]. Those five plugins were measured and named "frame collection", which aggregates frames into a batch using Gst-nvstreammux; "neural network application", which processes the batch through a neural model with Gst-nvinfer; "collage of frames", which creates a collage image from processed frames using Gst-nvmultistreamtiler; "post-inference processing", which

handles post-inference tasks like color conversion using Gst-nvvideoconvert; and "tracking objects", which performs visual tracking by drawing bounding boxes using Gst-nvdsosd. All of these data pipeline functions were traced and visualized. The second version of visualization shows the execution time for the whole iteration of the data pipeline. Instead of showing the execution time of separate processes within the pipeline, the GUI visualizes the execution runtime for a single batch of frames to be processed. The visualization module is implemented by using Websockets and JavaScript, and it displays delay results dynamically, allowing for immediate feedback across multiple hardware platforms. This flexibility ensures that the tool can be used in various environments, making it an essential feature for real-time performance monitoring and analysis.

Figure 13 presents the results of the delay measurement visualized through the graphical user interface of the delay visualization module. The interface is structured as a horizontal bar chart, where the vertical axis lists the hardware units being monitored, and the horizontal axis represents time, with each unit marking one millisecond. The time axis is formatted as "HH:MM:ss.mmm", indicating hours, minutes, seconds, and milliseconds. The chart displays five types of colored horizontal bars, each representing a distinct component of the monitoring process within the DeepStream pipeline with respect to the legend at the top of the graph, which identifies these components: red for "frame collection", blue for "inference", yellow for "collage frames", green for "post processing", and purple for "tracking objects." These bars provide a visual representation of the delay associated with each process. The width of each bar corresponds to the duration of the process it represents, with wider bars indicating longer delays and narrower bars indicating shorter processing times. Each bar is defined by two key parameters: the entry point (start time) and the exit point (end time), both marked by precise timestamps. For instance, a bar with a starting timestamp of 02:14:03.505 and an ending timestamp of 02:14:03.511 would span six milliseconds on the time axis, representing the delay incurred during that process. This visualization allows users to quickly assess the performance of different components within the pipeline and identify potential bottlenecks based on the length of the delays. In summary, Figure 12 provides a detailed, time-based visualization of delay measurements across multiple hardware units, making it a valuable tool for analyzing and optimizing the performance of real-time data-processing systems.

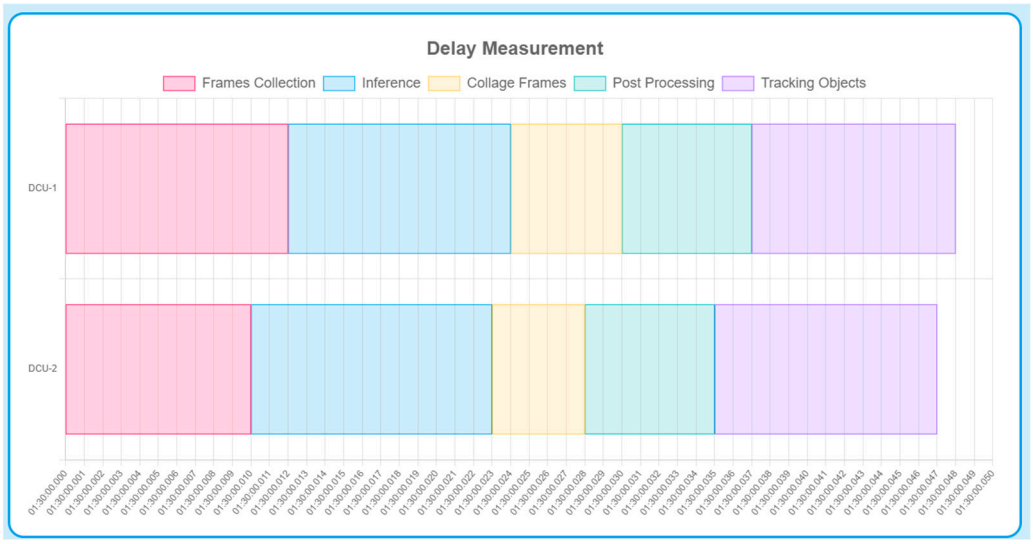

Figure 13. Task delay visualization GUI.

The visualization chart depicted in Figure 14 shows another version of the visualization GUI of our proposed tool. The figure shows the delay of the entire processing of a single

frame, where a single bar is single pipeline process iteration. The GUI is designed to display data-processing delays handled by designated units DCU-1 and DCU-2. Each of these processes is visually represented by a horizontal bar on the chart, where the length of the bar is directly proportional to the duration of the pipeline executing a single frame, effectively plotted against a time scale on the x-axis. Embedded within this visualization are several dynamic features designed to enhance user interaction and data analysis efficacy. A prominent feature is the adjustable limit indicator, a stark red vertical line within each bar, denoting a predefined threshold limit of 50 ms. This threshold is not static; users can dynamically modify it using an input box and a "Set Limit" button strategically placed adjacent to the chart. This interactivity allows users to adapt the threshold based on evolving real-time requirements or operational benchmarks, thereby tailoring the analysis to current system performance. Moreover, the bars on the chart change color based on whether the duration of the task stays within or exceeds the adjusted limit. Bars that adhere to the set limit retain their original coloration, while those surpassing the limit transition to a deeper shade of red, providing a vivid visual cue of performance lag. This feature is crucial for immediately identifying tasks that compromise system efficiency and may require further investigation or optimization. Integrating these responsive data interaction capabilities allows the visualization tool not only to present comprehensive data but also to enable dynamic analysis. This adaptability makes it an invaluable asset for performance management in environments where efficient real-time data processing is crucial, such as in network operations centers or during complex data-intensive operations. The tool's ability to provide immediate visual feedback on system performance, coupled with its interactive features, enhances operational oversight and supports proactive management of computational resources.

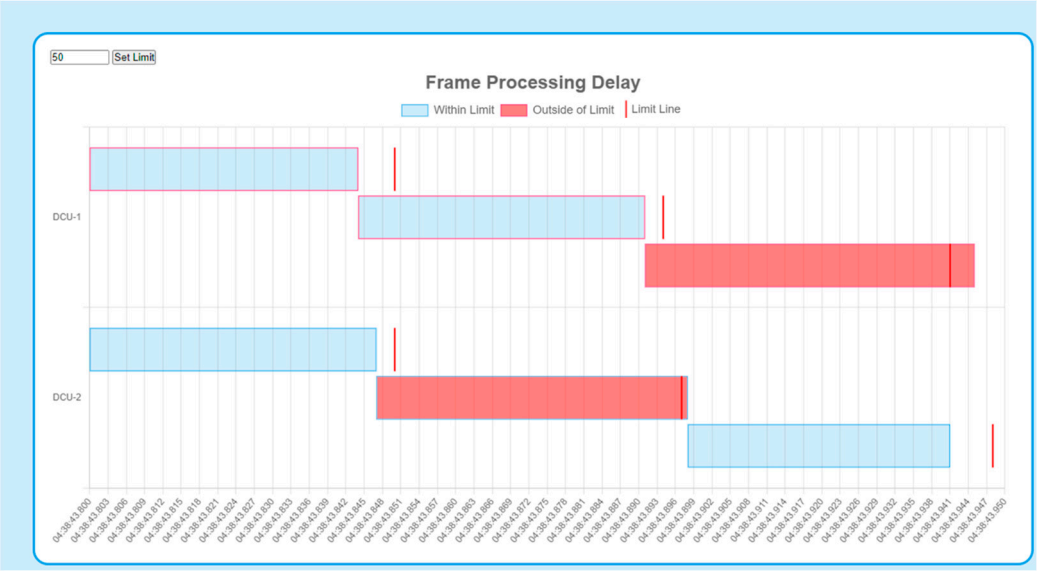

Figure 14. Frame-processing delay visualization GUI.

5. Conclusions

In this paper, we proposed the delay measurement tool to measure and visualize latency of AI analysis processes and identify bottlenecks in data pipelines. The design of the proposed tool has three software modules, consisting of the software tracing module, the delay calculation module, and the delay visualization module. The software tracing module captures and records the execution of processes within the system in real time, which provides essential data for subsequent delay calculations by logging precise timestamps and process sequences. The delay calculation module filters and organizes the

trace data into meaningful metrics, such as timestamps and delays, which are essential for identifying bottlenecks with respect to processing or components in data analysis applications. The delay visualization module plays an important role in the proposed tool, translating complex delay data into intuitive graphical representations that allow users to easily interpret and respond to latency issues. The proposed delay measurement tool addresses critical challenges in autonomous systems and distributed applications, particularly in autonomous vehicle (AV) navigation within urban settings. Synchronized video streams are essential for detecting and tracking moving objects, such as vehicles and pedestrians, in real time. Delays in these streams can cause temporal misalignments, leading to inaccuracies in object recognition or trajectory prediction. By measuring delays at critical pipeline stages, the tool helps identify bottlenecks, align data streams, and reduce latency, enhancing the AV's ability to make accurate, split-second decisions in dynamic environments. Beyond navigation, the tool is valuable for managing sensor failures or interruptions in distributed systems. For instance, in AVs equipped with multiple data concentrator units (DCUs), synchronization delays or failures can disrupt the system, especially during high-speed maneuvers. The tool can detect such issues in real time and trigger fault-tolerant strategies, such as activating backup sensors or redistributing tasks, ensuring operational continuity. Its adaptability extends to robotics, where real-time performance is critical for tasks like object manipulation, and to smart infrastructure systems, such as distributed sensor networks in smart cities. By ensuring synchronized data streams, it can improve traffic flow, pedestrian safety, and urban responsiveness. While this study focuses on synchronized video streams, the modular design allows for future integration with other sensor modalities, like LiDAR and radar, enabling broader applications in AV systems, robotics, and smart cities. The results obtained from the proposed delay measurement tool provide valuable insights into latency and bottlenecks within AI-powered data pipelines, offering practical applications for improving autonomous vehicle systems. First, the tool allows for pinpointing delays or bottlenecks at specific pipeline components, which can then be further analyzed and addressed to enhance system performance. By identifying critical sections where delays occur, the tool facilitates targeted improvements to reduce latency and optimize responsiveness. Additionally, the tool provides a baseline performance analysis, serving as a reference for testing and validating future optimization strategies or system upgrades. These delay insights also offer guidance for resource tuning, enabling the redistribution of computational workloads across available hardware to balance system efficiency. Furthermore, the tool supports continuous performance monitoring over time, allowing developers to identify patterns or trends in system behavior that may affect long-term performance. A notable outcome is the potential for automatic parameter adjustments based on identified bottlenecks, such as dynamically prioritizing specific sensor data or tuning system configurations to ensure timely processing and responsiveness. Lastly, the tool lays the foundation for future automation strategies, such as integrating AI-driven resource scheduling or dynamic optimizations to further streamline system performance. Collectively, these capabilities demonstrate the versatility of the delay measurement tool in addressing critical performance challenges, supporting ongoing system refinement, and enhancing the responsiveness and reliability of autonomous vehicle operations.

Our experiment results showed that the tool is effective in handling a variety of data types and scenarios, providing clear insights into the impact of different operational parameters on system delays. We were able to verify the delay in the object recognition experiment scenario by increasing the number of cameras and the number of video files as follows. The results showed that the total average delay per video (ADV) for video file data at 60 FPS with a bit rate of 7955 kbps is considerably higher by 4.45 ms when compared to the same frame rate at 4971 kbps, and 5.45 ms higher compared to 3982 kbps. Video

stream data at 60 FPS consistently exhibit a lower total average delay, reaching 131.67 ms, compared to the 121.67 ms seen with video file data at the same bitrate of 3982 kbps. Overall, it is clear that the ADV of all types of scenarios increases as the number of inputs increases. These results highlight the pronounced effect of higher bitrates on processing delays and effect of the input number and type.

The proposed delay measurement tool, while currently focused on synchronized video data streams, is inherently modular and adaptable to other sensor modalities commonly used in autonomous systems, such as LiDAR and radar. These sensors introduce unique challenges due to their asynchronous data acquisition and distinct data formats. For instance, LiDAR generates high-frequency, three-dimensional spatial data, often requiring precise synchronization with other sensors, while radar typically operates with varying sampling rates and outputs data streams with lower spatial resolution but higher reliability under adverse conditions. The tool's reliance on synchronized timestamps provides a strong foundation for handling such modalities. Through extensions to its preprocessing modules, the tool can incorporate dynamic time alignment techniques, such as timestamp interpolation and real-time clock drift correction, to synchronize asynchronous data streams. Additionally, the visualization module can be customized to accommodate the unique characteristics of LiDAR and radar data, providing insights into delay patterns and bottlenecks specific to these sensors. Future work will focus on integrating these modalities into the tool's framework and conducting tests with both synthetic and real-world datasets. This will enable comprehensive evaluations of the tool's performance in heterogeneous, multi-sensor environments, further validating its applicability to autonomous systems and distributed infrastructures.

While the proposed delay measurement tool demonstrates significant potential for synchronized multi-node environments, several limitations highlight areas for future improvement. First, the current study focuses on a dual-node configuration, serving as a proof of concept. Although this approach effectively demonstrates high availability and synchronization, it does not address the challenges of scalability in larger systems. Extending the tool to accommodate more nodes may introduce synchronization overhead, increased computational demands, and potential network bottlenecks, all of which require further optimization. Additionally, the study predominantly evaluates video data streams, limiting its applicability to other sensor modalities commonly used in autonomous systems, such as LiDAR, radar, and audio. These data types present unique characteristics, including higher data rates and asynchronous sampling, that could complicate synchronization and delay measurement. Moreover, the reliance on synchronized cameras operating at fixed frame rates (30 FPS and 60 FPS) constrains the tool's applicability to systems with variable or asynchronous frame rates, which are often encountered in real-world scenarios. Another limitation is the potential overhead introduced by real-time monitoring, which, while minimized in the current design, may still pose challenges in resource-constrained environments, affecting overall system performance.

Future work will focus on addressing these limitations to enhance the tool's scalability, versatility, and efficiency. To support larger multi-node systems, advanced synchronization protocols such as hierarchical PTP or clock drift compensation mechanisms will be explored. Expanding the tool's scope to integrate additional sensor modalities will enable comprehensive evaluations of multi-sensor autonomous systems. Moreover, incorporating support for asynchronous data streams and dynamic frame rates will enhance its adaptability to diverse real-world conditions. Efforts will also be directed towards optimizing the tool's resource utilization by leveraging lightweight tracing mechanisms or hardware-assisted profiling to reduce computational overhead. Finally, enhancing the visualization module with features such as real-time bottleneck detection, predictive analytics, and adaptive parameter tuning

will improve usability and provide actionable insights for system optimization. These advancements aim to ensure the tool's applicability to a broader range of scenarios while maintaining its accuracy and efficiency in delay measurement.

Author Contributions: Writing—original draft, A.Y.; writing—review and editing, S.P.; supervision, J.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National IT Industry Promotion Agency (NIPA) by the Korean government (MSIT) (S1503-24-1002) and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (Artificial Intelligence Graduate School Program (GIST)) (No. 2019-0-01842).

Data Availability Statement: The data used to support the findings of this study are included within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fleetwood, J. Public health, ethics, and autonomous vehicles. *Am. J. Public Health* **2017**, *107*, 532–537. [CrossRef] [PubMed]
2. Pandharipande, A.; Cheng, C.; Dauwels, J.; Gurbuz, S.Z.; Ibanez-Guzman, J.; Li, G.; Piazzoni, A.; Wang, P.; Santra, A. Sensing and machine learning for automotive perception: A review. *IEEE Sens. J.* **2023**, *23*, 11097–11115. [CrossRef]
3. Liu, L.; Liu, S.; Zhong, R.; Wu, B.; Yao, Y.; Zhang, Q.; Shi, W. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet Things J.* **2021**, *8*, 6469–6486. [CrossRef]
4. Yeong, D.J.; Velasco-Hernandez, G.; Barry, J.; Walsh, J. Sensor and sensor fusion technology in autonomous vehicles: A review. *Sensors* **2021**, *21*, 2140. [CrossRef]
5. Anvarjon, Y.; Park, S.; Kim, J. Design and implementation of data concentrator unit supported with multiple synchronized cameras for object-detection. In Proceedings of the 2023 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), Busan, Republic of Korea, 1–2 June 2023; pp. 1–2. [CrossRef]
6. Campbell, S.; O'Mahony, N.; Krpalkova, L.; Riordan, D.; Walsh, J.; Murphy, A.; Ryan, C. Sensor technology in autonomous vehicles: A review. In Proceedings of the 2018 29th Irish Signals and Systems Conference (ISSC), Belfast, UK, 21–22 June 2018; pp. 1–4. [CrossRef]
7. Giacalone, J.-P.; Bourgeois, L.; Ancora, A. Challenges in aggregation of heterogeneous sensors for autonomous driving systems. In Proceedings of the 2019 IEEE Sensors Applications Symposium (SAS), Sophia Antipolis, France, 11–13 March 2019; pp. 1–5. [CrossRef]
8. Gu, J.; Lind, A.; Chhetri, T.R.; Bellone, M.; Sell, R. End-to-end multimodal sensor dataset collection framework for autonomous vehicles. *Sensors* **2023**, *23*, 6783. [CrossRef] [PubMed]
9. Weidendorfer, J. Sequential performance analysis with Callgrind and KCachegrind. *Tools High Perform. Comput.* **2008**, *4939*, 93–113. [CrossRef]
10. Cantrill, B.M.; Shapiro, M.; Leventhal, A.H. Dynamic instrumentation of production systems. In Proceedings of the 2004 USENIX Annual Technical Conference, Boston, MA, USA, 27 June–2 July 2004.
11. Jia, J.; Lin, X.; Lin, F.; Liu, Y. DCU-CHK: Checkpointing for large-scale CPU-DCU heterogeneous computing systems. *CCF Trans. High Perform. Comput.* **2024**, *6*, 519–532. [CrossRef]
12. Li, Z.; Hasegawa, A.; Azumi, T. Autoware_Perf: A tracing and performance analysis framework for ROS 2 applications. *J. Syst. Archit.* **2021**, *123*, 102341. [CrossRef]
13. Havelund, K.; Roşu, G. An Overview of the Runtime Verification Tool Java PathExplorer. *Form. Methods Syst. Des.* **2004**, *24*, 189–215. [CrossRef]
14. Chen, F.; Rosu, G. Java-MOP: A monitoring-oriented programming environment for Java. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3440, pp. 546–550. [CrossRef]
15. Paxson, V. Bro: A system for detecting network intruders in real time. In Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, 26–29 January 1999.
16. Pellizzoni, R.; Meredith, P.; Caccamo, M.; Rosu, G. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In Proceedings of the Real-Time System Symposium (RTSS'08), Barcelona, Spain, 30 November–3 December 2008; pp. 481–491. [CrossRef]
17. Bédard, C.; Lütkebohle, I.; Dagenais, M. ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ROS 2. *arXiv* **2022**, arXiv:2201.00393. [CrossRef]

18. Las-Casas, P.; Mace, J.; Guedes, D.; Fonseca, R. Weighted sampling of execution traces: Capturing more needles and less hay. *ACM SIGCOMM Comput. Commun. Rev.* **2018**, *48*, 326–332. [CrossRef]
19. Desnoyers, M.; Dagenais, M. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In Proceedings of the Ottawa Linux Symposium (OLS), Ottawa, ON, Canada, 19–22 July 2006.
20. Mertz, J.; Nunes, I. Tigris: A DSL and framework for monitoring software systems at runtime. *arXiv* **2021**, arXiv:2103.15986. [CrossRef]
21. Kong, S.; Lu, M.; Li, L.; Gao, L. "Runtime Monitoring of Software Execution Trace: Method and Tools. *IEEE Access* **2020**, *8*, 114020–114036. [CrossRef]
22. Mertz, J.; Nunes, I. Software runtime monitoring with adaptive sampling rate. *J. Syst. Softw.* **2023**, *202*, 111708. [CrossRef]
23. Yusupov, A.; Park, S.; Kim, J.W. Data concentrator unit supported with intelligent video analytical data pipeline for autonomous vehicles. In Proceedings of the 9th International Conference on Advanced Engineering and ICT-Convergence, Jeju Island, Republic of Korea, 13–15 July 2022; pp. 264–268.
24. Park, S.; Ku, D.H.; Yusupov, A.; Kim, J.W. Design of cloud-native edge-based software framework for AI-DCU employing hybrid-V2X connectivity. In Proceedings of the 11th International Conference on Advanced Engineering and ICT-Convergence, Jeju Island, Republic of Korea, 12–14 July 2023; pp. 1–4.
25. Park, S.; Ji, K.H.; Ku, D.H.; Yusupov, A.; Kim, J.W. Design of virtual road driving test environment with driving simulator and DCU for bad weather SiLS data collection and verification. In Proceedings of the 10th International Conference on Advanced Engineering and ICT-Convergence, Bangkok, Thailand, 7–10 February 2023; pp. 101–105.
26. Wang, W.; Guo, K.; Cao, W.; Zhu, H.; Nan, J.; Yu, L. Review of electrical and electronic architectures for autonomous vehicles: Topologies, networking, and simulators. *Automot. Innov.* **2024**, *7*, 82–101. [CrossRef]
27. Wang, W.; Deng, H.; Sun, M.; Pan, Z. A cloud-connected autonomous driving system. In Proceedings of the 2020 IEEE 5th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA), Chengdu, China, 25–27 April 2020; pp. 96–102. [CrossRef]
28. Alparslan, O.; Arakawa, S.; Murata, M. Next generation intra-vehicle backbone network architectures. In Proceedings of the 2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR), Paris, France, 7–9 June 2021; pp. 1–7. [CrossRef]
29. Nair, A.G.; Seema, P.N.; Nair, M.G. Gateway DCU for backbone network communication architecture in a vehicle or in LAN network. In Proceedings of the 2023 World Conference on Communication & Computing (WCONF), Raipur, India, 15–17 February 2023; pp. 1–10. [CrossRef]
30. Velasco-Hernandez, G.; Yeong, D.J.; Barry, J.; Walsh, J. Autonomous driving architectures, perception and data fusion: A review. In Proceedings of the 2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP), Cluj-Napoca, Romania, 3–5 September 2020; pp. 315–321. [CrossRef]
31. Hu, H.; Wu, J.; Xiong, Z. A soft time synchronization framework for multi-sensors in autonomous localization and navigation. In Proceedings of the 2018 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM), Auckland, New Zealand, 9–12 July 2018; pp. 694–699. [CrossRef]
32. NVIDIA Corporation. DeepStream SDK 6.2. Available online: https://developer.nvidia.com/deepstream-sdk (accessed on 12 September 2024).
33. EfficiOS Inc. Babeltrace 2 (Version 2.0). Available online: https://babeltrace.org/ (accessed on 12 September 2024).
34. Websockets (Version 10.3). Available online: https://websockets.readthedocs.io/en/stable/ (accessed on 12 September 2024).
35. Precision Time Protocol (PTP) (Version 3.1). Available online: https://linuxptp.sourceforge.net/ (accessed on 12 September 2024).
36. High-Availability Concept. TechTarget. Available online: https://www.techtarget.com/searchdatacenter/definition/high-availability (accessed on 12 September 2024).
37. phc2sys (Clock Synchronization). Linux PTP. Available online: https://linuxptp.nwtime.org/documentation/phc2sys/ (accessed on 12 September 2024).
38. Milan, A.; Leal-Taixé, L.; Reid, I.D.; Roth, S.; Schindler, K. Mot16: A benchmark for multi-object tracking. *arXiv* **2016**, arXiv:1603.00831.
39. Dendorfer, P.; Osep, A.; Milan, A.; Schindler, K.; Cremers, D.; Reid, I.; Roth, S.; Leal-Taixé, L. Motchallenge: A benchmark for single-camera multiple target tracking. *Int. J. Comput. Vis.* **2021**, *129*, 845–881. [CrossRef]
40. Tomar, S. Converting video formats with FFmpeg. *Linux J.* **2006**, *146*, 10.
41. Wiseman, Y. Video compression prototype for autonomous vehicles. *Smart Cities* **2024**, *7*, 758–771. [CrossRef]
42. YUV Image Format. Videobolt. Available online: https://videobolt.net/motion-graphics-glossary/yuv-color-space (accessed on 12 September 2024).

43. UYVY Format. The Imaging Source. Available online: https://www.theimagingsource.com/en-us/documentation/icimagingcontrolcpp/PixelformatUYVY.htm (accessed on 12 September 2024).
44. GStreamer. GStreamer: Open Source Multimedia Framework. Available online: https://gstreamer.freedesktop.org/ (accessed on 12 September 2024).